Stochastic GPA Descent Report

- 1004318 Bryan Tan (Chen Zhengyu)
- 1005185 Varshini Harthachitramalogan
- 1004865 Wang Yanbao
- 1004875 Xiang Siqi
- 1005330 Christy Lau Jin Yun

Summary

Models Considered

- **Logistic Regression**
- **MultinomialNB**
- **XGBoost**
- **Extra Trees Classifier**
- **VotingClassifier**

Other libraries used

- **TPOT**: Python Automated Machine Learning tool that optimizes machine learning pipelines using genetic programming. Can output pipelines that combines/stack multiples models + preprocessors in its default state.
- **Optuna**: Hyperparameter tuning framework which can find the optimal machine learning in a large search space comparable to that of Random Grid Search, but with smarter choices of hyperparameters in each iteration and faster rate of model evaluation, ultimately leading to faster convergence.

Approaches to model training/evaluation

- Evaluate all the above models using Stratified 3 Fold Cross Validation; most of them are optimised using Optuna
- Utilise TPOT to search for machine learning pipelines that was not previously considered.

How we arrived at final model

- ExtraTreesClassifier performed well on public dataset.
- Pipelines output from TPOT also performed well on public dataset.
- In order to prevent overfitting and improve model performance through the use of independent classifiers, we created a Voting Classifier with soft voting from a few classifier pipelines from TPOT (which we called `cl0`, `cl2` and `cl12`) with an Extra Trees Classifier (which we called `etc`).
- We then used Optuna to optimise the weights assigned to each of the above classifiers.

Models Considered (Details)

Logistic Regression

What is the model?

- **Logistic Regression** is a linear model that outputs a probability (between 0 and 1) using the logistic function.

Why we chose to test this model?

- We use this model to provide a baseline f1_score which subsequent models ought to improve on.

Main Hyperparameters of the model

- **penalty** {'l1', 'l2', 'elasticnet', 'none'}, default='l2'
  Specifies the norm of the penalty
- **tol:** float, default = 1e-4
  Tolerance for stopping criteria.
- **C:** float, default = 1.0
  Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

Parameters tuning

- None; we used the default value provided by Sci-kit learn

Result

F1 score of Model on train-test split (75-25%): **0.6752**.

Multinomial NB

What is the model?

- **Multinomial NB** implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes. ### Why we chose to test this model?
  - The reason behind testing this model is based on the research paper released regarding the classification of hate speech 'A comparison of classification algorithms for hate speech detection'. The results show that the Multinomial Naive Bayes algorithm produces the best model with the highest recall value of **93.2%** which has an accuracy value of **71.2%** for the classification of hate speech. (**Putri et al., 2020**).

Hyperparameters of the model

- **alpha:** float, default = 1.0
  Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
- **fit_prior:** bool, default = True
  Whether to learn class prior probabilities or not. If false, a uniform prior will be used.
- **class_prior:** array-like of shape (n_classes,), default = None
  Prior probabilities of the classes. If specified, the priors are not adjusted according to the data

Parameters tuning

- We utilized the **Optuna** hyperparameter optimization framework to tune our hyperparameters.

Why we chose this framework

- Optuna boasts the following features:
  - **Lightweight, versatile, and platform agnostic architecture**
    - Handle a wide variety of tasks with a simple installation that has few requirements.
      - **Pythonic search spaces**
      - Define search spaces using familiar Python syntax including conditionals and loops.
      - **Efficient optimization algorithms**
      - Adopt state-of-the-art algorithms for sampling hyperparameters and efficiently pruning unpromising trials.
      - **Easy parallelization**
      - Scale studies to tens or hundreds or workers with little or no changes to the code.
      - **Quick visualization**
      - Inspect optimization histories from a variety of plotting functions.
- Optuna also allowed us to find the optimal parameters at a faster rate as opposed to grid search though there is a trade off in accuracy.

Result

F1 score of Model on public dataset (20%): **0.7100**

XGBoost

What is the model?

XGBoost is a **tree-based ensemble model** which stands for "gradient-boosted decision tree (GBDT)". Compared to the random forest, XGBoost implement the idea of "boosting" which establishes a connection between trees. Therefore, trees in XGBoost models are no longer independent of each other and the model eventually becomes an orderly collective decision-making system.

Why we chose to test this model?

XGBoost is an ensemble algorithm that has higher predicting power and performance, and it is achieved by improvisation on Gradient Boosting framework by introducing some accurate approximation algorithms. We speculate that by **gradient boost mechanism** and **self-regularization** embedded in the model can potentially increase the accuracy of predictions and reduce the risk of overfitting for tree-based models.

Hyperparameters of the model

- **eta:** float, default = 0.3
  The step size in fitting.
- **objective:** default = reg:squarederror
  Specify the learning task.
- **num_round:** integer
  Same as "n_estimators", the number for boosting, which also decides the number of trees in the ensemble forest.
- **subsample:** float, default = 1
  Subsample ratio of the training instances (prevent overfitting).
- **min_child_weight:** float, default = 1
  Minimum sum of instance weight needed in a child. The larger, the more conservative the model will be.
- **max_depth:** integer, default = 6
  Maximum depth of a tree.
- **gamma:** float, default = 0
  Minimum loss reduction required to make a further partition on a leaf node.
- **colsample_bytree:** default = 1
  The subsample ratio of columns when constructing each tree.

Parameters tuning

We are utilizing the **RandomSearchCV** provided by scikit-learn to tune our parameters.

Why we chose this framework

- RandomSearchCV is useful when we have **many parameters** to try and the training time is very long. The **training time** for XGBoost is relatively long compared to non-tree-based models. Considering that cross-validation takes a longer time as well, the train load is even heavier.
- The **number of parameters** to consider for XGBoost trees is particularly high and the magnitudes of influence are imbalanced. Compared to GridSearch, RandomSearch is more suitable in this situation.

Result

F1 score of Model on public dataset (20%): **0.6607**

Learning points

- **Overfitting problem:** It was shown during the training process that the f1-score on the training dataset was much higher than on the testing dataset under cross-validation (over 15% higher throughout the training process). Sometimes it happened that test loss didn't change while training loss was decreasing. Most **tree-based** models are easily overfitted. This means that increasing model complexity does not always increase model performance/robustness; **it's always about finding the balance between simplicity and complexity**.

Extra Tree Classifier (Included in the final Voting Classifier)

What is the model?

It is an ensemble machine learning model that is derived from deciesion trees. Also known as "Extremely Randomized Trees".

Why we chose to test this model?

ExtraTreesClassifier builds multiple trees and fits a number of randomized decision trees on various sub-samples of the dataset, averaging them to improve predictive accuracy and control over-fitting.

Hyperparameters:

- **n_estimators:** int, default = 100
  The number of trees in the forest.
- **criterion:** {"gini", "entropy", "log_loss"}, default = "gini"
  The function to measure the quality of a split.
- **max_depth:** int, default = None
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split:** int or float, default = 2
  The minimum number of samples required to split an internal node. If int, then consider min_samples_split as the minimum number. If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.
- **min_samples_leaf:** int or float, default = 1
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches.
- **min_weight_fraction_leaf:** float, default = 0.0
  The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- **max_features:** {"sqrt", "log2", None}, int or float, default = "sqrt"
  e number of features to consider when looking for the best split.If int, then consider max_features features at each split.

If float, then max_features is a fraction and round(max_features * n_features) features are considered at each split. If "auto", then max_features=sqrt(n_features). If "sqrt", then max_features=sqrt(n_features). If "log2", then max_features=log2(n_features). If None, then max_features=n_features.

- **max_leaf_nodes:** int, default = None
  Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.
- **min_impurity_decrease:** float, default = 0.0
  A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
- **bootstrap:** bool, default=False
  Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.
- **random_state:** int, RandomState instance or None, default=None
  Seed to control sources of randomness
- **class_weight:** {"balanced", "balanced_subsample"}, dict or list of dicts, default=None
  Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one.
- **ccp_alpha:** non-negative float, default=0.0
  Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed.
- **max_samples:** int or float, default=None
  If bootstrap is True, the number of samples to draw from X to train each base estimator.

Parameters tuning

- We utilized the **Optuna** hyperparameter optimization framework to tune our hyperparameters.

Parameters obtained

```
model = ExtraTreesClassifier(n_estimators=144,
                             max_depth=589,
                             criterion="entropy",
                             class_weight="balanced_subsample",
                             ccp_alpha=6.267622143679782e-05,
                             min_samples_split=157,
                             min_weight_fraction_leaf=4.8022857076483334e-05,
                             min_impurity_decrease=1.5576259402879695e-05,
                             max_features=0.00502175457189458,
                             max_samples=0.8999810323985775,
                             bootstrap=True)
```

Results

F1 score of Model on public dataset (20%) = **0.72273**

Out of the models evaluated above, Extra Trees Classifier performed the best. However, in addition to testing single models, we also attempted to use a library to generate pipelines that perform well with the given dataset. This was done using TPOT.

TPOT: An pipeline-generating AutoML library using genetic programming

What is TPOT ?

- **TPOT** stands for Tree-based Pipeline Optimization Tool. It is a Python Automated Machine Learning tool that optimizes machine learning pipelines using genetic programming

Why we chose to use TPOT ?

- TPOT automates the most tedious part of machine learning by intelligently exploring thousands of possible pipelines to find the best one the given data.
- It is built on scikit-learn (familiar interface)

  #### Key parameters of TPOT
  - **generations:** int or None, optional. default = 100
    Number of iterations to the run pipeline optimization process.

- **population_size:** nt, optional (default=100)
  Number of individuals to retain in the genetic programming population every generation. Must be a positive number.
- **scoring:** string or callable, optional (default='accuracy')
  Function used to evaluate the quality of a given pipeline for the classification problem. The following built-in scoring functions can be used:
- **cv:** int, cross-validation generator, or an iterable, optional (default=5)
  Cross-validation strategy used when evaluating pipelines.
- **config_dict:** Python dictionary, string, or None, optional (default=None)
  A configuration dictionary for customizing the operators and parameters that TPOT searches in the optimization process.

- **random_state:** int, RandomState instance or None, default=None
  he seed of the pseudo random number generator used in TPOT.

Implementation:

```
import random
from tpot import TPOTClassifier

print("reading CSVs")
train_df = pd.read_csv("/kaggle/input/50007-dataset/train_tfidf_features.csv")
test_df = pd.read_csv("/kaggle/input/50007-dataset/test_tfidf_features.csv")

print("creating X and y")
X = train_df.drop(['id', 'label'], axis=1).values
y = train_df['label'].values
```

▶ (click to expand) searchDict =

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=3)

periodic_checkpoint_folder = "/kaggle/working/checkpoints/"
model = TPOTClassifier(generations=500, population_size=30, cv=cv, config_dict=searchDict,
                       scoring='f1_macro', verbosity=3, random_state=random.randint(1, 99999999
9),
                       n_jobs=2, periodic_checkpoint_folder=periodic_checkpoint_folder,
                       max_time_mins=700, max_eval_time_mins=15)
# perform the search
model.fit(X, y)
# export the best model
model.export('tpot_best_model.py')
```

Running the above implmentation on several kaggle notebooks for 12 hours each yield various well-performing cross-validated pipelines. The three selected pipelines are as follows:

```
from sklearn.pipeline import make_pipeline, make_union
from sklearn.preprocessing import Normalizer, FunctionTransformer, RobustScaler, MaxAbsScaler, Bi
narizer, MinMaxScaler
from sklearn.feature_selection import VarianceThreshold, SelectPercentile, f_classif
from tpot.export_utils import set_param_recursive
from tpot.builtins import StackingEstimator
from sklearn.naive_bayes import BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import SGDClassifier
from copy import copy

# Average CV score on the training set was: 0.7165325962375743
cl0 = make_pipeline(
    make_union(
        FunctionTransformer(copy),
        make_pipeline(
            make_union(
                Normalizer(norm="max"),
                SelectPercentile(score_func=f_classif, percentile=5)
            ),
            MaxAbsScaler()
        )
    ),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_rati
o=0.5,
                                              learning_rate="invscaling", loss="modified_huber",
 penalty="elasticnet", power_t=0.5)),
    BernoulliNB(alpha=1.0, fit_prior=True)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl0.steps, 'random_state', 2)


# Average CV score on the training set was: 0.7167734749214567
cl2 = make_pipeline(
    StackingEstimator(estimator=DecisionTreeClassifier(
        criterion="gini", max_depth=8, min_samples_leaf=9, min_samples_split=6)),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_rati
o=0.75,
                                              learning_rate="invscaling", loss="perceptron", pena
lty="elasticnet", power_t=0.1)),
    BernoulliNB(alpha=1.0, fit_prior=False)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl2.steps, 'random_state', 222)


# Average CV score on the training set was: 0.7192365888770997
cl12 = make_pipeline(
    SelectPercentile(score_func=f_classif, percentile=77),
    SelectPercentile(score_func=f_classif, percentile=68),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=0.0012, fit_intercept=False,
                                              l1_ratio=0.0, learning_rate="invscaling", loss="hin
ge", penalty="elasticnet", power_t=0.5)),
    BernoulliNB(alpha=2.300000000000001, fit_prior=False)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl12.steps, 'random_state', 422)
```

Because there is an extremely large variety of preprocessors and hyperparameters defined above, we will exclude the explaination of the preprocessors and only attempt to explain the main models used in the above 3 pipelines

Which models are included?

1. SGD Classifier:

Explanation:

- implements a plain stochastic gradient descent learning routine which supports different loss functions and penalties for classification. Below is the decision boundary of a SGDClassifier trained with the hinge loss, equivalent to a linear SVM.

Hyperparameters:

- **penalty:** {'l2', 'l1', 'elasticnet'}, default='l2'
  The penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.
- **alpha:** float, default=0.0001
  Constant that multiplies the regularization term. The higher the value, the stronger the regularization. Also used to compute the learning rate when set to learning_rate is set to 'optimal'. Values must be in the range [0.0, inf).
- **l1_ratio:** float, default=0.15
  The Elastic Net mixing parameter, with 0 <= l1_ratio <= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1. Only used if penalty is 'elasticnet'. Values must be in the range [0.0, 1.0].
- **fit_intercept:** bool, default=True
  Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.
- **max_iter:** int, default=1000
  The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the fit method, and not the partial_fit method. Values must be in the range [1, inf).

- **tol:** float, default=1e-3
  The stopping criterion. If it is not None, training will stop when (loss > best_loss - tol) for n_iter_no_change consecutive epochs. Convergence is checked against the training loss or the validation loss depending on the early_stopping parameter. Values must be in the range [0.0, inf).

2. Decision Tree Classifier:

Explanation:

- A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

Hyperparameters:

- **criterion:** {"gini", "entropy", "log_loss"}, default="gini"
  The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see Mathematical formulation.
- **splitter:** {"best", "random"}, default="best"
  The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
- **max_depth:** int, default=None
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split:** int or float, default=2
  The minimum number of samples required to split an internal node:
  If int, then consider min_samples_split as the minimum number.
  If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

- **min_samples_leaf:** int or float, default=1
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
  If int, then consider min_samples_leaf as the minimum number.
  If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

- **min_weight_fraction_leaf:** float default=0.0
  The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- **max_features:** int or float {"auto", "sqrt", "log2"}, default=None
  The number of features to consider when looking for the best split:
  If int, then consider max_features features at each split.
  If float, then max_features is a fraction and max(1, int(max_features * n_features*in*)) features are considered at each split.
  If "auto", then max_features=sqrt(n_features).
  If "sqrt", then max_features=sqrt(n_features).
  If "log2", then max_features=log2(n_features).
  If None, then max_features=n_features.

3. Bernoulli Naive Bayes

Explanation:

- Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable.

Hyperparameters:

- **alpha:** float, default=1.0
  Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
- **binarize:** float or None, default=0.0
  Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.
- **fit_prior:** bool, default=True
  Whether to learn class prior probabilities or not. If false, a uniform prior will be used.
- **class_prior:** array-like of shape (n_classes,), default=None
  Prior probabilities of the classes. If specified, the priors are not adjusted according to the data.

FINAL MODEL CREATION

In order to craft a model that is performant and resilient, we used a voting classifier to aggregate the predicted probabilties of the Extra Trees Classifier and 3 of the best performing TPOT pipelines into a single model.

The rationale is that a variety of model that is independent is able to resolve and adjust for the errors of any single model included within the ensemble

VotingClassifier

What is the model?

- **VotingClassifier** is a machine learning model that trains an ensemble of user-defined models and predicted an output based on the weights (voting power) assigned to each of the model.

Selected Models for the voting classifier

- **1. Stacking Classifier1: SGD Classifier and Bernoulli Naive Bayes**
  - **Preprocessing**:
    - Values are rescaled by the maximum of the absolute values
    - Select highest scoring percentage (5 percentile) of features
    - Scale each feature by its maximum absolute value
  - **SGD Classifier**
  - **Bernoulli Naive Bayes**
  - **Individual Performance on cross validation**: `0.7165325962375743`

Code for first model from TPOT

```
# Average CV score on the training set was: 0.7165325962375743
cl0 = make_pipeline(
    make_union(
        FunctionTransformer(copy),
        make_pipeline(
            make_union(
                Normalizer(norm="max"),
                SelectPercentile(score_func=f_classif, percentile=5)
            ),
            MaxAbsScaler()
        )
    ),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_rati
o=0.5,
                                              learning_rate="invscaling", loss="modified_huber",
 penalty="elasticnet", power_t=0.5)),
    BernoulliNB(alpha=1.0, fit_prior=True)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl0.steps, 'random_state', 2)
```

- **2. Stacking Classifier2: Decision Tree, SGD Classifier and Bernoulli Naive Bayes**
  - **Decision Tree**
  - **SGD classifier**
  - **Bernoulli Naive Bayes**
  - **Individual Performance**: `0.7167734749214567` **### Code for second model from TPOT**

```
# Average CV score on the training set was: 0.7167734749214567
cl2 = make_pipeline(
    StackingEstimator(estimator=DecisionTreeClassifier(
        criterion="gini", max_depth=8, min_samples_leaf=9, min_samples_split=6)),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_rati
o=0.75,
                                              learning_rate="invscaling", loss="perceptron", pena
lty="elasticnet", power_t=0.1)),
    BernoulliNB(alpha=1.0, fit_prior=False)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl2.steps, 'random_state', 222)
```

- **3. Stacking Classifier3: SGD Classifier and Bernoulli Naive Bayes**
  - **Preprocessing**:
    - Select highest scoring percentage (77 percentile) of features
    - Select highest scoring percentage (68 percentile) of features
  - **Decision Tree**
  - **SGD classifier**
  - **Bernoulli Naive Bayes**
  - **Individual Performance**: `0.7192365888770997` **### Code for last model from TPOT**

```
# Average CV score on the training set was: 0.7192365888770997
cl12 = make_pipeline(
    SelectPercentile(score_func=f_classif, percentile=77),
    SelectPercentile(score_func=f_classif, percentile=68),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=0.0012, fit_intercept=False,
                                              l1_ratio=0.0, learning_rate="invscaling", loss="hin
ge", penalty="elasticnet", power_t=0.5)),
    BernoulliNB(alpha=2.300000000000001, fit_prior=False)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl12.steps, 'random_state', 422)
```

- **4. Extra Tree Classifier**
  - **Individual Performance**: `0.72273`
  - **code for model 4**: **### Code for Optuna-optimised ExtraTreesClassifier**

```
etc = ExtraTreesClassifier(n_estimators=144,
                           max_depth=589,
                           criterion='entropy',
                           class_weight='balanced_subsample',
                           ccp_alpha=6.267622143679782e-05,
                           min_samples_split=157,
                           min_weight_fraction_leaf=4.8022857076483334e-05,
                           min_impurity_decrease=1.5576259402879695e-05,
                           max_features=0.00502175457189458,
                           max_samples=0.8999810323985775,
                           bootstrap=True)
```

Voting Classifier: Brief Explanation and Parameters

- **ExtraTreesClassifier** and **a mix of dissimilar models/pipelines** (3 pipelined stacking classifiers)
- The Reason for this approach is to ensure that ExtraTreesClassifier does not overfit too much to the public test set on kaggle.
- **ExtraTreesClassifier** had the highest score on the public test set on kaggle. However, we feared that it was overfitting to the public dataset, hence our inclusion of other models with ostensibly lower performance on the public test set.
- The TPOT models hence has a regularising effect on the overall model (we assess that the reduction in variance is worth the potential increase in bias)
- `voting = 'soft'`: Predicts the class label based on the argmax of the sums of the predicted probabilities
- `weight`: Multiple the predicted probability of each model with the assigned weight

```
estimatorsLast = [("cl0", cl0), ("cl2", cl2), ("cl12", cl12), ("etc", etc)]
w1 = 1
w2 = 2
w3 = 3.5
w4 = 10
weights = [w1, w2, w3, w4]
model = VotingClassifier(estimators=estimatorsLast,
                         voting='soft',
                         verbose=False,
                         n_jobs=2, weights=weights)
```

How were the above weights chosen? It was based on a mix of an initial parameter search by Optuna, followed by manual tuning.

What is optuna?

- Optuna is an automatic hyperparameter optimization software framework, particularly designed for machine learning. It features an imperative, define-by-run style user API. Optuna enjoys high modularity, and the user of Optuna can dynamically construct the search spaces for the hyperparameters.

Why we chose this framework? (RECAP)

- Optuna boasts the following features:
  - **Lightweight, versatile, and platform agnostic architecture**
    - Handle a wide variety of tasks with a simple installation that has few requirements.
      - **Pythonic search spaces**
      - Define search spaces using familiar Python syntax including conditionals and loops.
      - **Efficient optimization algorithms**
      - Adopt state-of-the-art algorithms for sampling hyperparameters and efficiently pruning unpromising trials.
      - **Easy parallelization**
      - Scale studies to tens or hundreds or workers with little or no changes to the code.
      - **Quick visualization**
      - Inspect optimization histories from a variety of plotting functions.

  Optuna also allowed us to find the optimal parameters at a faster rate as opposed to grid search though there is a trade off in accuracy.

Implementation:

```
import optuna
from sklearn.ensemble import VotingClassifier, ExtraTreesClassifier


def objective(trial):
    cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=1)

    ### modeling with suggested params
    w1 = trial.suggest_float("w1", 0.5, 6)
    w2 = trial.suggest_float("w2", 0.5, 6)
    w3 = trial.suggest_float("w3", 0.5, 6)
    w4 = trial.suggest_float("w4", 0.5, 6)

    weights = [w1, w2, w3, w4]

    # Average CV score on the training set was: 0.7165325962375743
    cl0 = make_pipeline(
        make_union(
            FunctionTransformer(copy),
            make_pipeline(
                make_union(
                    Normalizer(norm="max"),
                    SelectPercentile(score_func=f_classif, percentile=5)
                ),
                MaxAbsScaler()
            )
        ),
        StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_
ratio=0.5,
                                                  learning_rate="invscaling", loss="modified_hube
r", penalty="elasticnet", power_t=0.5)),
        BernoulliNB(alpha=1.0, fit_prior=True)
    )
    # Fix random state for all the steps in exported pipeline
    set_param_recursive(cl0.steps, 'random_state', 2)

    # Average CV score on the training set was: 0.7167734749214567
    cl2 = make_pipeline(
        StackingEstimator(estimator=DecisionTreeClassifier(
            criterion="gini", max_depth=8, min_samples_leaf=9, min_samples_split=6)),
        StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_
ratio=0.75,
                                                  learning_rate="invscaling", loss="perceptron",
 penalty="elasticnet", power_t=0.1)),
        BernoulliNB(alpha=1.0, fit_prior=False)
    )
    # Fix random state for all the steps in exported pipeline
    set_param_recursive(cl2.steps, 'random_state', 222)

    # Average CV score on the training set was: 0.7192365888770997
    cl12 = make_pipeline(
        SelectPercentile(score_func=f_classif, percentile=77),
        SelectPercentile(score_func=f_classif, percentile=68),
        StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=0.0012, fit_intercept=False,
                                                  l1_ratio=0.0, learning_rate="invscaling", loss
="hinge", penalty="elasticnet", power_t=0.5)),
        BernoulliNB(alpha=2.300000000000001, fit_prior=False)
    )
    # Fix random state for all the steps in exported pipeline
    set_param_recursive(cl12.steps, 'random_state', 422)

    etc = ExtraTreesClassifier(n_estimators=144,
                               max_depth=589,
                               criterion='entropy',
                               class_weight='balanced_subsample',
                               ccp_alpha=6.267622143679782e-05,
                               min_samples_split=157,
                               min_weight_fraction_leaf=4.8022857076483334e-05,
                               min_impurity_decrease=1.5576259402879695e-05,
                               max_features=0.00502175457189458,
                               max_samples=0.8999810323985775,
                               bootstrap=True)
    estimatorsLast = [("cl0", cl0), ("cl2", cl2), ("cl12", cl12), ("etc", etc)]

    model = VotingClassifier(estimators=estimatorsLast,
                             voting='soft',
                             verbose=False,
                             n_jobs=2, weights=weights)
    ## cross validation score
    score = cross_val_score(model, X, y, n_jobs=2, cv=cv, scoring="f1_macro")
    f1_mean = score.mean()

    return f1_mean


study = optuna.create_study(direction='maximize')  # maximize accuracy
study.optimize(objective, n_trials=None, timeout=42000, n_jobs=2,)
print(study.best_trial.params)
print(study.best_value)
```

Running the above Optuna search on a kaggle notebook for 12 hours resulted in the following weights:

```
{'w1': 0.5053634333272882,
 'w2': 0.8790582376550015,
 'w3': 1.4825640308089136,
 'w4': 5.669220731735056}
```

This was manually tuned to arrive at the final chosen weight:

```
w1 = 1
w2 = 2
w3 = 3.5
w4 = 10
```

FINAL MODEL USED

```
from sklearn.ensemble import VotingClassifier, ExtraTreesClassifier
from sklearn.pipeline import make_pipeline, make_union
from sklearn.preprocessing import Normalizer, FunctionTransformer, RobustScaler, MaxAbsScaler, Bi
narizer, MinMaxScaler
from sklearn.feature_selection import VarianceThreshold, SelectPercentile, f_classif
from tpot.export_utils import set_param_recursive
from tpot.builtins import StackingEstimator
from sklearn.naive_bayes import BernoulliNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import SGDClassifier
from copy import copy
from sklearn.model_selection import StratifiedKFold

cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=1)

# Weights chosen after Optuna hyperparamter tuning and manual adjustments
w1 = 1
w2 = 2
w3 = 3.5
w4 = 10
weights = [w1, w2, w3, w4]

cl0 = make_pipeline(
    make_union(
        FunctionTransformer(copy),
        make_pipeline(
            make_union(
                Normalizer(norm="max"),
                SelectPercentile(score_func=f_classif, percentile=5)
            ),
            MaxAbsScaler()
        )
    ),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_rati
o=0.5,
                                              learning_rate="invscaling", loss="modified_huber",
 penalty="elasticnet", power_t=0.5)),
    BernoulliNB(alpha=1.0, fit_prior=True)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl0.steps, 'random_state', 2)

# Average CV score on the training set was: 0.7167734749214567
cl2 = make_pipeline(
    StackingEstimator(estimator=DecisionTreeClassifier(
        criterion="gini", max_depth=8, min_samples_leaf=9, min_samples_split=6)),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=1.0, fit_intercept=False, l1_rati
o=0.75,
                                              learning_rate="invscaling", loss="perceptron", pena
lty="elasticnet", power_t=0.1)),
    BernoulliNB(alpha=1.0, fit_prior=False)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl2.steps, 'random_state', 222)

# Average CV score on the training set was: 0.7192365888770997
cl12 = make_pipeline(
    SelectPercentile(score_func=f_classif, percentile=77),
    SelectPercentile(score_func=f_classif, percentile=68),
    StackingEstimator(estimator=SGDClassifier(alpha=0.001, eta0=0.0012, fit_intercept=False,
                                              l1_ratio=0.0, learning_rate="invscaling", loss="hin
ge", penalty="elasticnet", power_t=0.5)),
    BernoulliNB(alpha=2.300000000000001, fit_prior=False)
)
# Fix random state for all the steps in exported pipeline
set_param_recursive(cl12.steps, 'random_state', 422)

etc = ExtraTreesClassifier(n_estimators=144,
                           max_depth=589,
                           criterion='entropy',
                           class_weight='balanced_subsample',
                           ccp_alpha=6.267622143679782e-05,
                           min_samples_split=157,
                           min_weight_fraction_leaf=4.8022857076483334e-05,
                           min_impurity_decrease=1.5576259402879695e-05,
                           max_features=0.00502175457189458,
                           max_samples=0.8999810323985775,
                           bootstrap=True)
estimatorsLast = [("cl0", cl0), ("cl2", cl2), ("cl12", cl12), ("etc", etc)]


model = VotingClassifier(estimators=estimatorsLast,
                         voting='soft',
                         verbose=False,
                         n_jobs=2, weights=weights)
```

How are most of our models evaluated?

Cross Validation

- To prevent potential overfitting issue, we use cross validation to evaluate the performance of our final model. We split the training dataset into **5 folds**, each time use **4** of them for training and **1** of them for validation. The labels of the dataset are also balanced (stratified) according to the ratio of positive and negative labels. After getting the result from all trials, we average the f1-score as our final evaluation matric.

Implementation:

```
from sklearn.model_selection import cross_val_score
## cross validation score
cv2 = cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=1)
score = cross_val_score(model, X, y, n_jobs=2, cv=cv2, scoring="f1_macro")

print("LATEST")
print(score)
print(score.mean())
print(score.std())
```

Analysis, Learning Points and Other considerations:

Increasing model complexity does not always increase model Performance

- Simple models such as Naive Bayes can perform surprisingly well and sometimes even outperform complex models such as XGBoost etc.
- An increase in model complexity increases the tendency of a model to overfitting, potentially leading to misleadingly high validation scores but poor test scores. Complex models also tend to take a longer time to train and test. Using them can be extremely resource intensive
- It is hence necessary that a good balanced between simplicity and complexity is found, and that cross-validation is performed to ensure robustness of the results.

Real-world datasets are large and requires substantial resources to utilise for ML

- Models, especially complex ones, can take a very long time to train on real-world datasets.
- This means that model size/performance is an important consideration in real-world scenarios, and that small models are highly preferable.
- Algorithms and preprocessors that perform dimensionality reduction/encoding of the dataset can increase model performance while decreasing the amount of time needed to train models. An in-depth practical and technical understanding of such preprocessors is hence extremely valuable.

PCA

- In our cases, it appears that many models' performance decreased after doing PCA on data. In this particular case, it might be because: the features were already filtered for one time (we only choose the first 5000 words after tf-idf processing), and most noises have been filtered. It also indicates one of the disadvantages of PCA which is the loss of information. However, it is also true that the training time was much shorter after decreasing the number of features. We need to be careful in PCA operations and most of the time the balance of time, performance and the property of the dataset are keys that decide how we utilize PCA. In this project, it is clear that this dataset does not contain too much noise and we aim to maximize the performance, therefore, we did not apply PCA to the data before training.

Stacking Models

- Even though almost all the models we used in task 3 have been taught in class, it is the first time we tried to stack them together. Compared to other ensemble techniques such bagging and boosting, stacking model is very different from them. Unlike bagging, in stacking, the models are typically different (e.g. not all decision trees) and fit on the same dataset (e.g. instead of samples of the training dataset). Unlike boosting, in stacking, a single model is used to learn how to best combine the predictions from the contributing models (e.g. instead of a sequence of models that correct the predictions of prior models). There are often 2 or more models responsible for fitting on the training data and predicting the result and one model for combine the predictions of these models.
  Since it is easy to implement and quite effective, we could consider to include this technique in our syllabus.

Extra Tree Classifier:

- Similar to random forest, extra tree classfier is another trees ensemble methods. While it is different from random forest since it adds more randomization and thus reduce bias and variance. In terms of computational cost, the extra trees algorithm is faster. Thus it provides another choice for us when we get stuck at increasing accuracy for random forest.

Techniques to fine tune the hyperparameters:

- AutoML + Hyper-parameter tuning with TPOT
- Optuna