SHOULD I FEATURE MY PRODUCT ON THIS NEWLY PUBLISHED MASHABLE ARTICLE?

INTRODUCTION

Context and problem

We're Design Engineers seeking to advertise our product on Mashable, a global multi-platform media and entertainment company with one of its focuses being Tech. We were featured during our product launch, but in order to maintain interest and engagement, so we aim to be included in the "featured articles" section at the bottom of an article. We accessed a Mashable dataset and realised that most articles get around 1000 shares with a small fraction becoming viral, getting from 10x as many shares up to 800x. Being featured in a random article corresponds to a 1/20 chance of it being viral, and this is not enough for us, which is why **we aim to develop a Machine Learning algorithm that analyses different attributes in newly published Mashable articles and determines if will become viral**. We want to detect virality as soon as possible to maximise exposure and get a good price.

Introduction to the dataset

Our dataset, *OnlineNewsPopularity,* is from the UCI Machine Learning Repository and contains features for 39797 articles published by Mashable from January 7, 2013 to January 7, 2015. Of its 61 attributes, 58 are predictive, 2 non-predictive and 1 is a goal field. The heterogeneous attributes extracted by the authors include:

- Non-predictive identifiers (URL…)
- Time relative attributes (Weekday_is_Monday…)
- Digital media atributes (Num_Imgs…)
- Word count attributes (N_Tokens_Title…)
- Features linked to previously published articles referenced in the new article(Num_Self_Hrefs…)
- Natural language processing features obtained through the Latent Dirichlet Allocation algorithm (Lda_00…)

The innovative aspect of this dataset is that it uses features available before article publication, effectively allowing the writers of a Mashable article to improve content before its released and to obtain predictors for the interaction it will receive. This could be useful not only for content creators but also for advertisement businesses and even the writing of political campaigns.

Very few articles exceed 10,000 shares and this will give us an indicator for setting the threshold. This plot shows our data is highly unbalanced, this will need fixing for the training set. The median is 1400, the mean 3395 and the standard dev. 11627.

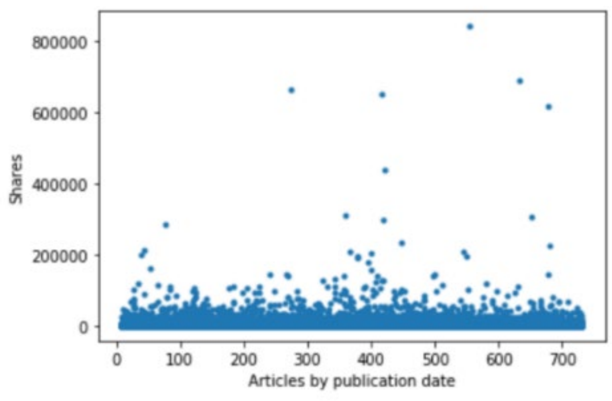


Figure 1: Plot showing the variation of shares

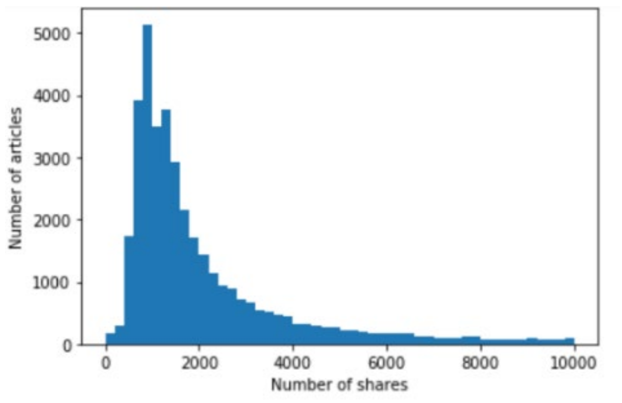


Figure 2: Histogram showing the distribution of shares below 10000

Data preparation

First, we had to binarize the target attribute, *shares.* We defined as viral the top 5% of data; that is, the ones exceeding 11,000 shares. With this we created a new column (*shares_b*) and assigned all data points a true/ false Boolean value.

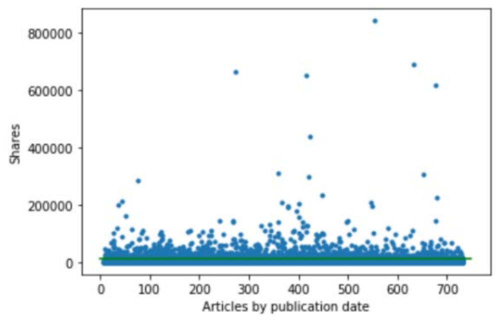


Figure 3: linear scale plot with threshold in green

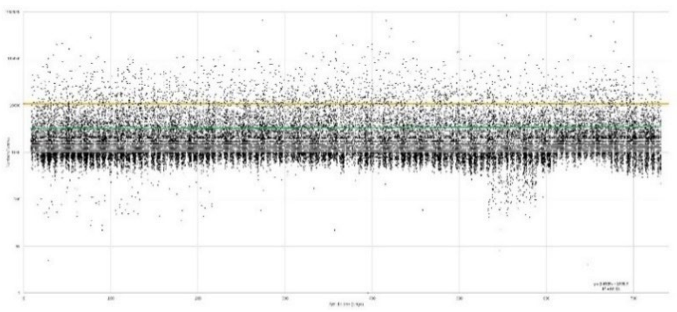


Figure 4: log scale plot with threshold value in green

Secondly, the following attributes were removed:

- *URL*: Because it is a non-predictive attribute.
- *timedelta*: We want to predict virality a short time after publication, so we don't want our model to be influenced by how long an article has been online. The flat yellow line on the Excel plot shows an absence of correlation between time delta and shares.
- *is_weekend*: This attribute is repeated as the same information is found in columns *Weekday_Is_Saturday* and *Weekday_Is_Sunday*.
- *shares*: after binarization through setting a threshold this column it no longer served a purpose in the prediction. The outcome will be Boolean (*shares_b*)

Then we split the data into three sets for training, testing and validation with them containing 80%, 10%, and 10% of the data points respectively. We left the validation and test sets untouched to ensure we could properly evaluate our models in a real-life situation. In order to balance our training data, we under-sampled the negative values for *shares_b* for they constitute the majority class and would bias our model. However, we also trained one model on imbalanced data to see if real world proportions where most articles aren't viral could better train our model. We also standardised the underdamped data.

Performance metrics in context

- **Accuracy:** The proportion of articles which have been correctly predicted to be popular or unpopular.
- **Precision:** The proportion of the articles predicted to be popular which turn out to be popular.
- **Recall**: The proportion of articles predicted to be unpopular which turn out to be popular.
- For a balanced data set, accuracy is a suitable metric to use as the proportion of correctly predicted popular and unpopular articles are equally weighted
- For an imbalanced data set, precision would be our most important metric as we want to ensure that if an article is predicted to be popular and we invest to be featured, that the article does turn out to be popular.

For our circumstances and low budget, want to ensure the article we choose to feature on will go viral. Considering our data set is imbalanced greatly towards the non-viral outcome, **we will optimise models for precision**. We don't mind waiting long for the viral article, as long as our chances of finding it are high and we don't invest money on a false positive.

THE PREDICTIVE MODELS

Linear regression and seaborn plot

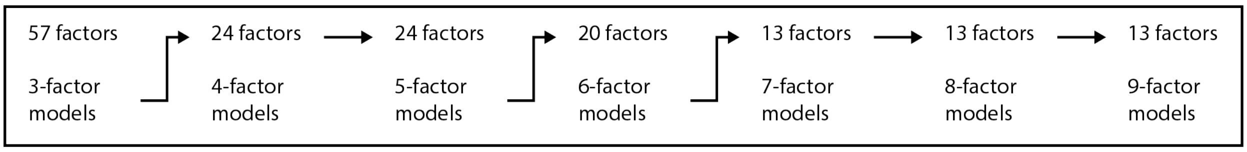
We carried out initial linear regression by looking at 2 factor correlation plotted in a seaborn heatmap (found in appendix). This gave us some initial insight into the possible relation between *shares_b* and *LDA_02*, *kw_avg_avg* and *num_href* as they appear in a lighter colour.

Logistic regression: forward selection

Algorithm design

A development of the forward selection algorithm was created to search more combinations of factors. The algorithm is designed for parallelisation, using the concurrent futures library in Python to utilise all CPU cores. Instead of starting with one factor and sequentially trying new factors to build a model, the algorithm generates all possible combinations of factors of a given length, processes the performance across all cores, and collects/sorts the results. The best models are saved, and then tested with the validation set. This allows us to eliminate all overfitting models, good on training but not validation.

A shortlist is created with factors which appear in the better models, as they are intuitively more robust predictors. This shortlist is used to generate longer models, and so on, narrowing in on a final model. This was the flow of iteration for the testing below.



Testing

Three-factor models were tested first. Every combination was produced and validated. This model is trained on imbalanced data and therefore uses precision as its key metric.

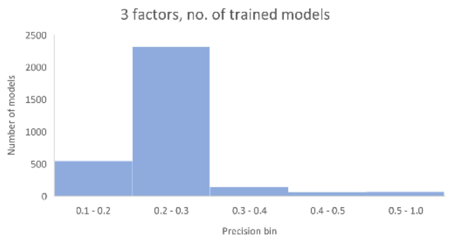


Figure 5

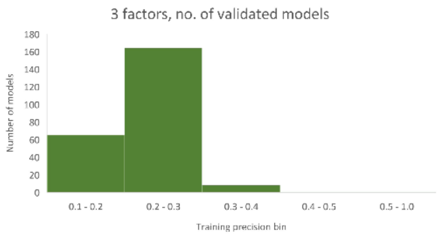


Figure 4

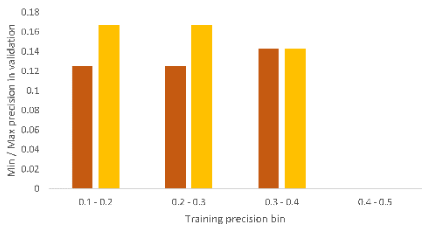


Figure 3 Min/max validation precision of 3 factor models grouped by training precision

Figure 5 shows the number of models generated in training, grouped by their precision score in 0.1 steps. Note how approximately 2200 models scored 0.2 - 0.3 precision, but only about 160 models of those models scored 0.2 - 0.3 in validation(Figure 6), these are the not over-fitted models.

Figure 7 shows within each training precision bin the best and worst precision score achieved in the validation set. Here we can see for the 0.2 - 0.3 training precision models, the best validation results were 0.1667. Since this is lower than the training result, the model may not be reliable. The model length is increased until the validation results are more similar to the training results, indicating a reliable model.

Up until the eight-factor models, training sets would record precisions over 0.3 but the validation set results would not exceed 0.25 showing poor consistency. However, there were four eight-factor models which scored precision over 0.3 in both validation and training sets (Figure 8, 0.3-0.4 training precision column). Of these four models the full results were obtained to show the best performing model. Its metrics are shown below:

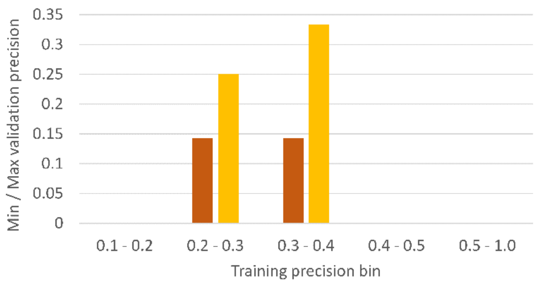


Figure 6 Min/max validation precision of eight-factor models grouped by training precision

Table 1 Final metrics of LogReg

	Final Eight Factor Model
Training Accuracy	0.950
Training Precision	0.375
Validation Accuracy	0.950
Validation Precision	0.333

Nonlinear Model

Decision Tree

To build our decision tree, we use the Gini Impurity – a measure of the probability that a randomly assigned datapoint is incorrect. The maximum Gini for our two classes is 0.5, whilst the minimum is 0. The algorithm is greedy making locally optimal decisions to end up near the global. In this case, the best decision is to pick the lowest weighted average Gini and continue from there, however it may not necessarily be the best model.

To ensure our model did not overfit and was optimal, we explored the use of two parameters that limited the number of variables, *max_depth* and *min_impurity decrease*. We firstly plotted *max_depth* versus *precision score* for the training and validation model with *min_impurity decrease* set to default 0 (pure tree).

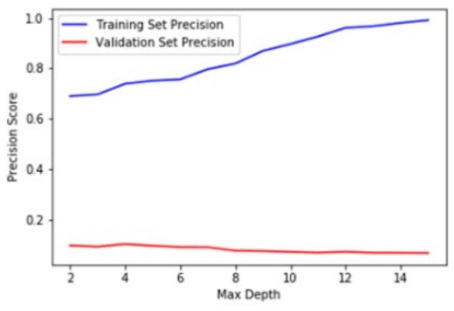


Figure 7

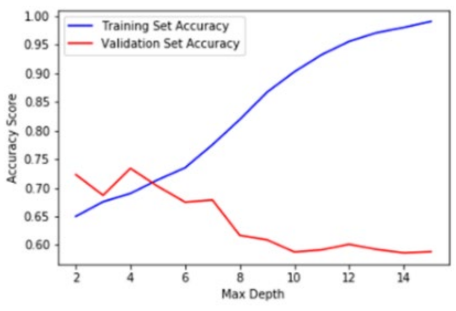


Figure 10

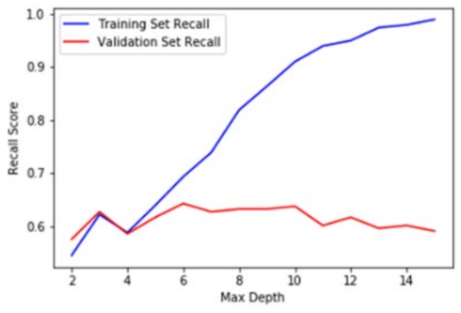


Figure 11

The results show that the validation data peaked at 4 features, and thereafter was overfitting the training data. As a result, this was used as the baseline for the next test. The weighted impurity decrease allows control over how deeply the tree grows based on Gini impurity, and allows us to define how separated classes must be to create new leaf nodes. The tree will stop when a new split would result in a decrease in Gini Impurity of less than the defined *min_impurity_decrease* - this again allows control on overfitting to choose the best model.

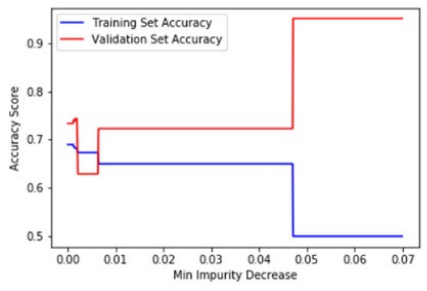


Figure 10

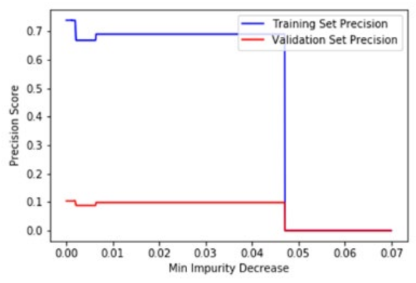


Figure 8

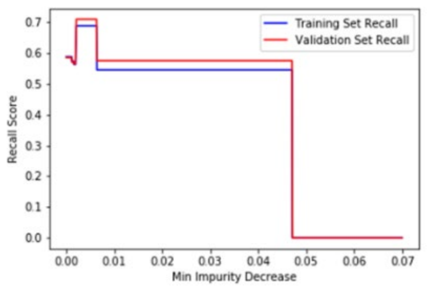


Figure 9

Figure 13 and 14 show that the highest precision and accuracy on the validation set was with a pure tree. The tree is visualised, and we end with 15 features. The first few splits had a high Gini of around 0.43, which is not ideal, but after this it reduced allowing for more distinct decisions that provided clear class definition. The precision for this model was acceptable at 10.405%, and the accuracy good, but the other model methods had better performance in comparison.

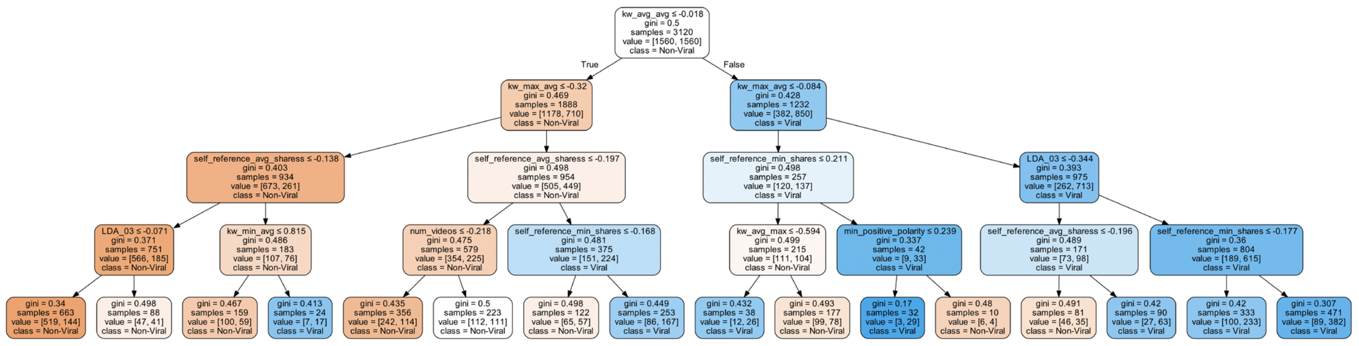

kw_avg_avg ≤ -0.018 gini = 0.5 samples = 3120 value = [1560, 1560] class = Non-Viral
True
False
kw_max_avg ≤ -0.32 gini = 0.469 samples = 1888 value = [1178, 710] class = Non-Viral
kw_max_avg ≤ -0.084 gini = 0.428 samples = 1232 value = [382, 850] class = Viral
self_reference_avg_sharess ≤ -0.138 gini = 0.403 samples = 934 value = [673, 261] class = Non-Viral
self_reference_avg_sharess ≤ -0.197 gini = 0.498 samples = 954 value = [505, 449] class = Non-Viral
self_reference_min_shares ≤ 0.211 gini = 0.498 samples = 257 value = [120, 137] class = Viral
LDA_03 ≤ -0.344 gini = 0.393 samples = 975 value = [262, 713] class = Viral
LDA_03 ≤ -0.071 gini = 0.371 samples = 751 value = [566, 185] class = Non-Viral
kw_min_avg ≤ 0.815 gini = 0.486 samples = 183 value = [107, 76] class = Non-Viral
num_videos ≤ -0.218 gini = 0.475 samples = 579 value = [354, 225] class = Non-Viral
self_reference_min_shares ≤ -0.168 gini = 0.481 samples = 375 value = [151, 224] class = Viral
kw_avg_max ≤ -0.594 gini = 0.499 samples = 215 value = [111, 104] class = Non-Viral
min_positive_polarity ≤ 0.239 gini = 0.337 samples = 42 value = [9, 33] class = Viral
self_reference_avg_sharess ≤ -0.196 gini = 0.489 samples = 171 value = [73, 98] class = Viral
self_reference_min_shares ≤ -0.177 gini = 0.36 samples = 804 value = [189, 615] class = Viral
gini = 0.34 samples = 663 value = [519, 144] class = Non-Viral
gini = 0.498 samples = 88 value = [47, 41] class = Non-Viral
gini = 0.467 samples = 159 value = [100, 59] class = Non-Viral
gini = 0.413 samples = 24 value = [7, 17] class = Viral
gini = 0.435 samples = 356 value = [242, 114] class = Non-Viral
gini = 0.5 samples = 223 value = [112, 111] class = Non-Viral
gini = 0.498 samples = 122 value = [65, 57] class = Non-Viral
gini = 0.449 samples = 253 value = [86, 167] class = Viral
gini = 0.432 samples = 38 value = [12, 26] class = Viral
gini = 0.493 samples = 177 value = [99, 78] class = Non-Viral
gini = 0.17 samples = 32 value = [3, 29] class = Viral
gini = 0.48 samples = 10 value = [6, 4] class = Non-Viral
gini = 0.491 samples = 81 value = [46, 35] class = Non-Viral
gini = 0.42 samples = 90 value = [27, 63] class = Viral
gini = 0.42 samples = 333 value = [100, 233] class = Viral
gini = 0.307 samples = 471 value = [89, 382] class = Viral

Figure 11 Final decision tree

Random Forest

This is an automated method of creating a non-linear model, it builds a number of similar decision trees that are not equal and uses them to find a prediction each, then uses majority voting to find the class predicted by most trees. You are thus trying a range of possibilities by making a lot of small variations in trees of the original dataset, then aggregating those to get the best result. This performed similarly to the Decision Tree model earlier, both with 15 features, but had a slightly lower precision and a lower accuracy. The features are different; however, the trees have similar Gini values when trying to split, confirming that the manually generated decision tree was an ideal outcome.

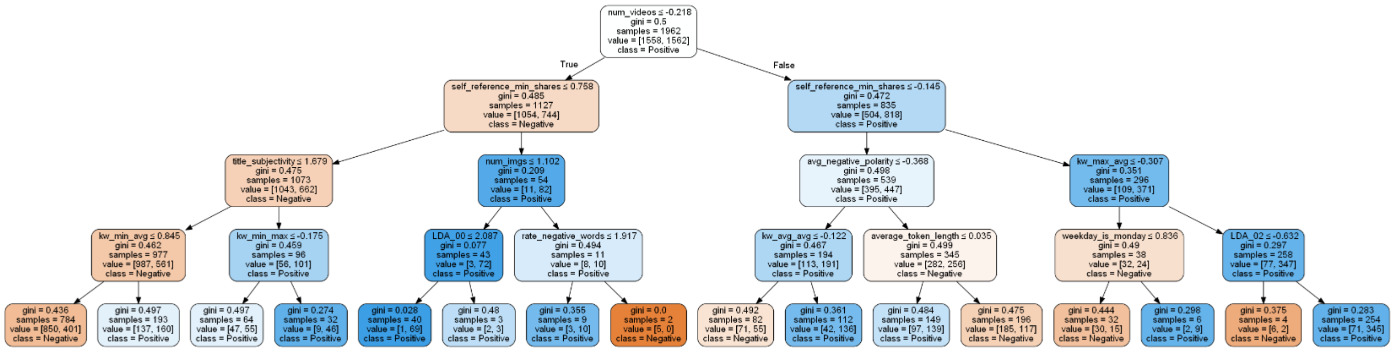

num_videos ≤ -0.218 gini = 0.5 samples = 1962 value = [1558, 1562] class = Positive
True
False
self_reference_min_shares ≤ 0.758 gini = 0.485 samples = 1127 value = [1054, 744] class = Negative
self_reference_min_shares ≤ -0.145 gini = 0.472 samples = 835 value = [504, 818] class = Positive
title_subjectivity ≤ 1.679 gini = 0.475 samples = 1073 value = [1043, 662] class = Negative
num_imgs ≤ 1.102 gini = 0.209 samples = 54 value = [11, 82] class = Positive
avg_negative_polarity ≤ -0.368 gini = 0.498 samples = 539 value = [395, 447] class = Positive
kw_max_avg ≤ -0.307 gini = 0.351 samples = 296 value = [109, 371] class = Positive
kw_min_avg ≤ 0.845 gini = 0.462 samples = 977 value = [987, 561] class = Negative
kw_min_max ≤ -0.175 gini = 0.459 samples = 96 value = [56, 101] class = Positive
LDA_00 ≤ 2.087 gini = 0.077 samples = 43 value = [3, 72] class = Positive
rate_negative_words ≤ 1.917 gini = 0.494 samples = 11 value = [8, 10] class = Positive
kw_avg_avg ≤ -0.122 gini = 0.467 samples = 194 value = [113, 191] class = Positive
average_token_length ≤ 0.035 gini = 0.499 samples = 345 value = [282, 256] class = Negative
weekday_is_monday ≤ 0.836 gini = 0.49 samples = 38 value = [32, 24] class = Negative
LDA_02 ≤ -0.632 gini = 0.297 samples = 258 value = [77, 347] class = Positive
gini = 0.436 samples = 784 value = [850, 401] class = Negative
gini = 0.497 samples = 193 value = [137, 160] class = Positive
gini = 0.497 samples = 64 value = [47, 55] class = Positive
gini = 0.274 samples = 32 value = [9, 46] class = Positive
gini = 0.028 samples = 40 value = [1, 69] class = Positive
gini = 0.48 samples = 3 value = [2, 3] class = Positive
gini = 0.355 samples = 9 value = [3, 10] class = Positive
gini = 0.0 samples = 2 value = [5, 0] class = Negative
gini = 0.492 samples = 82 value = [71, 55] class = Negative
gini = 0.361 samples = 112 value = [42, 136] class = Positive
gini = 0.484 samples = 149 value = [97, 139] class = Positive
gini = 0.475 samples = 196 value = [185, 117] class = Negative
gini = 0.444 samples = 32 value = [30, 15] class = Negative
gini = 0.298 samples = 6 value = [2, 9] class = Positive
gini = 0.375 samples = 4 value = [6, 2] class = Negative
gini = 0.283 samples = 254 value = [71, 345] class = Positive

Figure 12 Example tree from random forest

Support Vector Machine

In this case, the dataset is projected to a higher dimensional space to find the best hyperplane to split the data – and this is to better find the best separation between classes. For our context, we found a C value of 1 (no penalty), a 'poly' kernel and 'auto' gamma which is defined by 1/number of features. This had a high accuracy and comparable precision but provided a poor recall meaning that false negatives were high. In this case SVM is not desirable to use for our purpose of predicting viral articles. Again, the data was standardised before doing this test.

MODEL COMPARISON OF RESULTS, DISCUSSION

	Performance metrics on validation set				
Model	**Precision**	**Recall**	**Accuracy**	**Days until positive**	**Positive per day**
LogReg Algorithm	33%	0. 518%	95%	24	0.042
Decision Tree	10.405%	58.549%	73.382%	0.066	15
Random Forest	10.214%	66.839%	69.717%	0.055	18
Support Vector Machine	11.308%	26.425%	86.299%	0.16	6

We created a new metric based on the number of days it takes for the model to predict viral. This is based on the average number of publications per day and the proportion of articles the model predicts positive. The metric *days until positive* can be calculated using:

$$Ap = \frac{TP + FP}{As} * Ad$$

$Ap = $ *Positive predicted articles per day*
$Ad = $ *articles published per day*
$As = $ *number of articles in sample*
$TP, FP = $ *True positive, False positive*

It was reassuring to find the highest scoring attributes were common to most models. Amongst the most used attributes were kw_avg_avg , num_hrefs and num_imgs. The relevance of this attributes is not only recurring but also logical, as many other page rank algorithms are based on number of references and keywords to boost site searches.

CONCLUSION

Bearing in mind the performance matrix we have chosen the LogReg model for several reasons:

- **Highest precision:** This means we have the best chance of correctly predicting if an article will go viral, at the cost of having to wait several days due to the low recall.
- **Highest accuracy:** due to its very low false positives count. This is useful because we cant waste money on non-viral articles.
- **Simplicity:** a LogReg model is easier to interpret, providing a cleared insight into the impact of each factor on the outcome.

The LogReg model was used on the test set, predicting one viral out of the 3957 articles with 100% precision. This shows the conservative, but precise, nature of our model. Its accuracy was 95% and recall 0.5%. Its clear limitation is that in this case it would take 72 days to predict a positive article, due to the large number of viral articles that it misses for its low recall.

We highlight the SVM also performs well, providing a much higher recall which can come in handy if immediacy is ever required (for example if we were told to choose one of the articles published on a given day).

APPENDICES

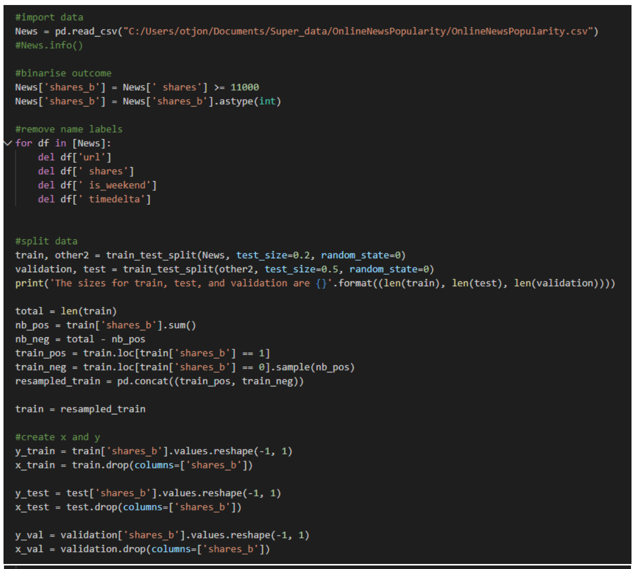
The data was obtained from https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity, where it had been donated by the authors of the first paper that analysed it :

K. Fernández, P. Vinagre and P. Cortez. A Proactive Intelligent Decision Support System for Predicting the Popularity of Online News. Proceedings of the 17th EPIA 2015 - Portuguese Conference on Artificial Intelligence, September, Coimbra, Portugal.

A. ATTRIBUTES TABLE

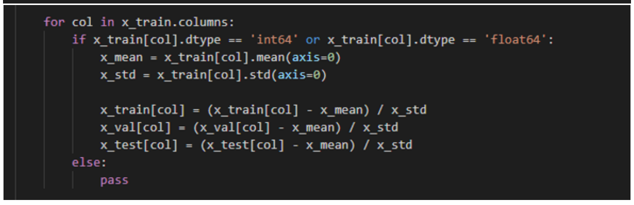
N_Tokens_Title	Number Of Words In The Title	**weekday_is_monday**	Was the article published on a Monday?
N_Tokens_Content	Number Of Words In The Content	**weekday_is_tuesday**	Was the article published on a Tuesday?
N_Unique_Tokens	Rate Of Unique Words In The Content	**weekday_is_wednesday**	Was the article published on a Wednesday?
N_Non_Stop_Words	Rate Of Non-Stop Words In The Content	**weekday_is_thursday**	Was the article published on a Thursday?
N_Non_Stop_Unique_Tokens	Rate Of Unique Non-Stop Words In The Content	**weekday_is_friday**	Was the article published on a Friday?
Num_Hrefs	Number Of Links	**weekday_is_saturday**	Was the article published on a Saturday?
Num_Self_Hrefs	Number Of Links To Other Articles Published By Mashable	**weekday_is_sunday**	Was the article published on a Sunday?
Num_Imgs	Number Of Images	**LDA_00**	Closeness to LDA topic 0
Num_Videos	Number Of Videos	**LDA_01**	Closeness to LDA topic 1
Average_Token_Length	Average Length Of The Words In The Content	**LDA_02**	Closeness to LDA topic 2
Num_Keywords	Number Of Keywords In The Metadata	**LDA_03**	Closeness to LDA topic 3
Data_Channel_Is_Lifestyle	Is Data Channel 'Lifestyle'?	**LDA_04**	Closeness to LDA topic 4
Data_Channel_Is_Entertainment	Is Data Channel 'Entertainment'?	**global_subjectivity**	Text subjectivity
Data_Channel_Is_Bus	Is Data Channel 'Business'?	**global_sentiment_polarity**	Text sentiment polarity
Data_Channel_Is_Socmed	Is Data Channel 'Social Media'?	**global_rate_positive_words**	Rate of positive words in the content
Data_Channel_Is_Tech	Is Data Channel 'Tech'?	**global_rate_negative_words**	Rate of negative words in the content
Data_Channel_Is_World	Is Data Channel 'World'?	**rate_positive_words**	Rate of positive words among non-neutral tokens
Kw_Min_Min	Worst Keyword (Min. Shares)	**rate_negative_words**	Rate of negative words among non-neutral tokens
Kw_Max_Min	Worst Keyword (Max. Shares)	**avg_positive_polarity**	Avg. polarity of positive words
Kw_Avg_Min	Worst Keyword (Avg. Shares)	**min_positive_polarity**	Min. polarity of positive words
Kw_Min_Max	Worst Keyword (Min. Shares)	**max_positive_polarity**	Max. polarity of positive words
Kw_Max_Max	Best Keyword (Max. Shares)	**avg_negative_polarity**	Avg. polarity of negative words
Kw_Avg_Max	Best Keyword (Avg. Shares)	**min_negative_polarity**	Min. polarity of negative words
Kw_Min_Avg	Avg. Keyword (Min. Shares)	**max_negative_polarity**	Max. polarity of negative words
Kw_Max_Avg	Avg. Keyword (Max. Shares)	**title_subjectivity**	Title subjectivity
Kw_Avg_Avg	Avg. Keyword (Avg. Shares)	**title_sentiment_polarity**	Title polarity
Self_Reference_Min_Shares	Min. Shares Of Referenced Articles In Mashable	**abs_title_subjectivity**	Absolute subjectivity level
Self_Reference_Max_Shares	Max. Shares Of Referenced Articles In Mashable	**abs_title_sentiment_polarity**	Absolute polarity level
Self_Reference_Avg_Shares	Avg. Shares Of Referenced Articles In Mashable	**shares**	Number of shares (target)

B. PREPARING THE DATASET



```
#import data
News = pd.read_csv("C:/Users/otjon/Documents/Super_data/OnlineNewsPopularity/OnlineNewsPopularity.csv")
#News.info()

#binarise outcome
News['shares_b'] = News[' shares'] >= 11000
News['shares_b'] = News['shares_b'].astype(int)

#remove name labels
for df in [News]:
    del df['url']
    del df[' shares']
    del df[' is_weekend']
    del df[' timedelta']


#split data
train, other2 = train_test_split(News, test_size=0.2, random_state=0)
validation, test = train_test_split(other2, test_size=0.5, random_state=0)
print('The sizes for train, test, and validation are {}'.format((len(train), len(test), len(validation))))

total = len(train)
nb_pos = train['shares_b'].sum()
nb_neg = total - nb_pos
train_pos = train.loc[train['shares_b'] == 1]
train_neg = train.loc[train['shares_b'] == 0].sample(nb_pos)
resampled_train = pd.concat((train_pos, train_neg))

train = resampled_train

#create x and y
y_train = train['shares_b'].values.reshape(-1, 1)
x_train = train.drop(columns=['shares_b'])

y_test = test['shares_b'].values.reshape(-1, 1)
x_test = test.drop(columns=['shares_b'])

y_val = validation['shares_b'].values.reshape(-1, 1)
x_val = validation.drop(columns=['shares_b'])
```

1. Binarising the target attribute: *shares*
2. Removing non-predictive and repeated attributes
3. Separating into training, validation and test sets.

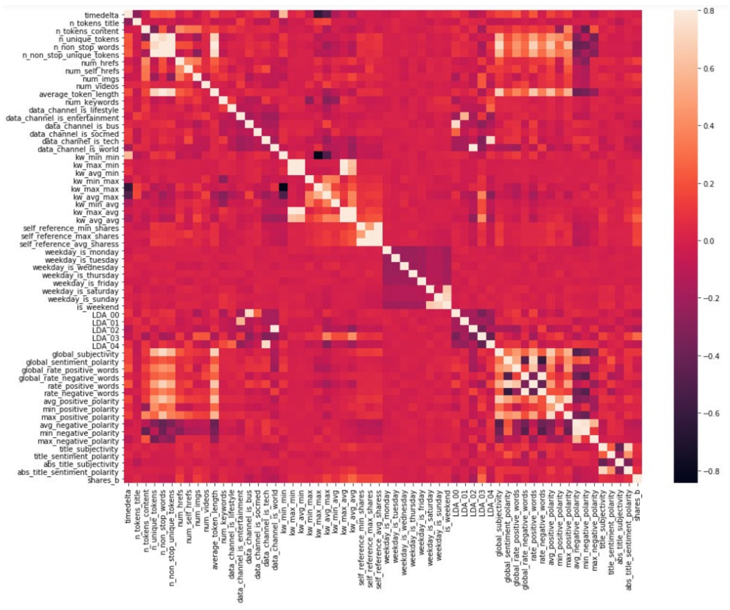


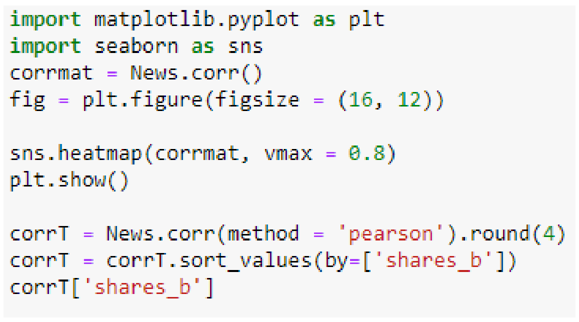
```
    for col in x_train.columns:
        if x_train[col].dtype == 'int64' or x_train[col].dtype == 'float64':
            x_mean = x_train[col].mean(axis=0)
            x_std = x_train[col].std(axis=0)

            x_train[col] = (x_train[col] - x_mean) / x_std
            x_val[col] = (x_val[col] - x_mean) / x_std
            x_test[col] = (x_test[col] - x_mean) / x_std
        else:
            pass
```

4. Standardisation of under-sampled training data

C. CODE FOR THE MODELS

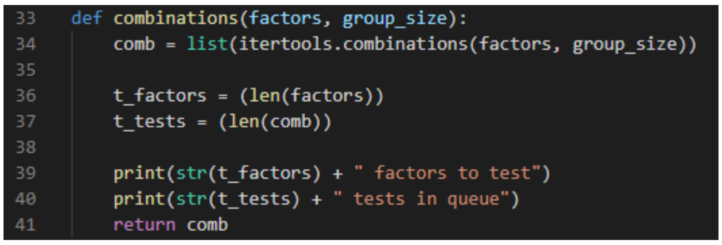
1. Seaborn Correlation Heatmap – for initial understanding of data and relationship of columns with *shares_b*





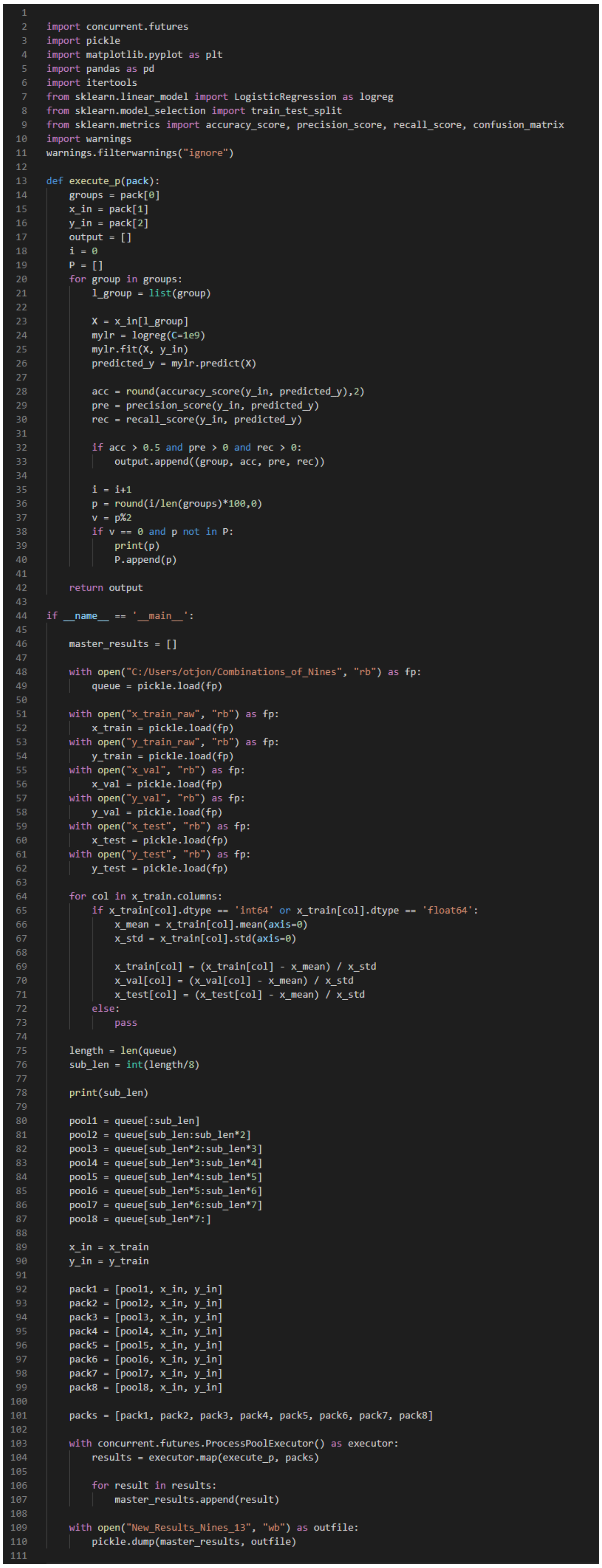
```
import matplotlib.pyplot as plt
import seaborn as sns
corrmat = News.corr()
fig = plt.figure(figsize = (16, 12))

sns.heatmap(corrmat, vmax = 0.8)
plt.show()

corrT = News.corr(method = 'pearson').round(4)
corrT = corrT.sort_values(by=['shares_b'])
corrT['shares_b']
```

2. LogReg main code



```
def combinations(factors, group_size):
    comb = list(itertools.combinations(factors, group_size))

    t_factors = (len(factors))
    t_tests = (len(comb))

    print(str(t_factors) + " factors to test")
    print(str(t_tests) + " tests in queue")
    return comb
```

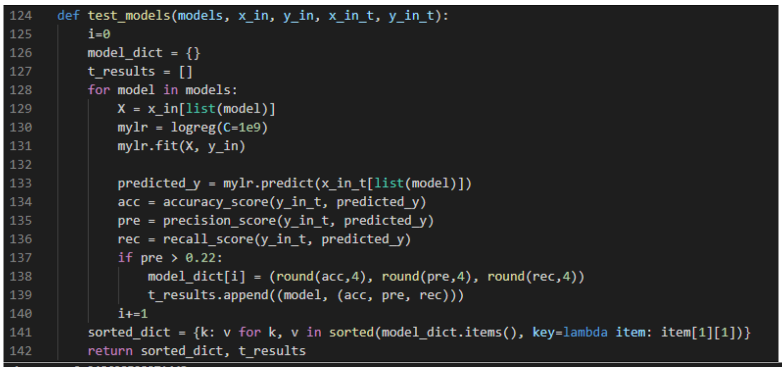
2.1 This function generates every possible combination of factors from the input list *factors* of length *group_size.*



```python

import concurrent.futures
import pickle
import matplotlib.pyplot as plt
import pandas as pd
import itertools
from sklearn.linear_model import LogisticRegression as logreg
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, confusion_matrix
import warnings
warnings.filterwarnings("ignore")

def execute_p(pack):
    groups = pack[0]
    x_in = pack[1]
    y_in = pack[2]
    output = []
    i = 0
    P = []
    for group in groups:
        l_group = list(group)

        X = x_in[l_group]
        mylr = logreg(C=1e9)
        mylr.fit(X, y_in)
        predicted_y = mylr.predict(X)

        acc = round(accuracy_score(y_in, predicted_y),2)
        pre = precision_score(y_in, predicted_y)
        rec = recall_score(y_in, predicted_y)

        if acc > 0.5 and pre > 0 and rec > 0:
            output.append((group, acc, pre, rec))

        i = i+1
        p = round(i/len(groups)*100,0)
        v = p%2
        if v == 0 and p not in P:
            print(p)
            P.append(p)

    return output

if __name__ == '__main__':

    master_results = []

    with open("C:/Users/otjon/Combinations_of_Nines", "rb") as fp:
        queue = pickle.load(fp)

    with open("x_train_raw", "rb") as fp:
        x_train = pickle.load(fp)
    with open("y_train_raw", "rb") as fp:
        y_train = pickle.load(fp)
    with open("x_val", "rb") as fp:
        x_val = pickle.load(fp)
    with open("y_val", "rb") as fp:
        y_val = pickle.load(fp)
    with open("x_test", "rb") as fp:
        x_test = pickle.load(fp)
    with open("y_test", "rb") as fp:
        y_test = pickle.load(fp)

    for col in x_train.columns:
        if x_train[col].dtype == 'int64' or x_train[col].dtype == 'float64':
            x_mean = x_train[col].mean(axis=0)
            x_std = x_train[col].std(axis=0)

            x_train[col] = (x_train[col] - x_mean) / x_std
            x_val[col] = (x_val[col] - x_mean) / x_std
            x_test[col] = (x_test[col] - x_mean) / x_std
        else:
            pass

    length = len(queue)
    sub_len = int(length/8)

    print(sub_len)

    pool1 = queue[:sub_len]
    pool2 = queue[sub_len:sub_len*2]
    pool3 = queue[sub_len*2:sub_len*3]
    pool4 = queue[sub_len*3:sub_len*4]
    pool5 = queue[sub_len*4:sub_len*5]
    pool6 = queue[sub_len*5:sub_len*6]
    pool7 = queue[sub_len*6:sub_len*7]
    pool8 = queue[sub_len*7:]

    x_in = x_train
    y_in = y_train

    pack1 = [pool1, x_in, y_in]
    pack2 = [pool2, x_in, y_in]
    pack3 = [pool3, x_in, y_in]
    pack4 = [pool4, x_in, y_in]
    pack5 = [pool5, x_in, y_in]
    pack6 = [pool6, x_in, y_in]
    pack7 = [pool7, x_in, y_in]
    pack8 = [pool8, x_in, y_in]

    packs = [pack1, pack2, pack3, pack4, pack5, pack6, pack7, pack8]

    with concurrent.futures.ProcessPoolExecutor() as executor:
        results = executor.map(execute_p, packs)

        for result in results:
            master_results.append(result)

    with open("New_Results_Nines_13", "wb") as outfile:
        pickle.dump(master_results, outfile)

```

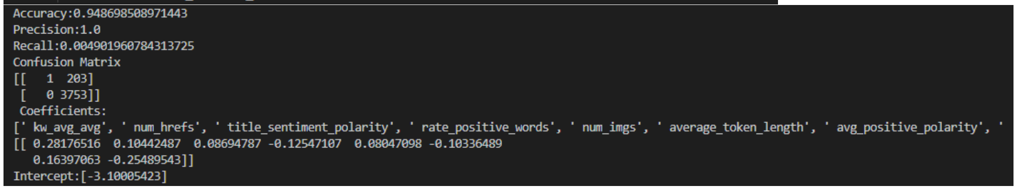
2.2 Function *execute_p* calculates the accuracy, precision and recall for every combination of factors entered.

The code in "*if __name__== '__main__'* "distributes the combinations generated previously into as many pools as processers are available. The function is then run on all CPU cores (multiprocessing).



```python
def test_models(models, x_in, y_in, x_in_t, y_in_t):
    i=0
    model_dict = {}
    t_results = []
    for model in models:
        X = x_in[list(model)]
        mylr = logreg(C=1e9)
        mylr.fit(X, y_in)

        predicted_y = mylr.predict(x_in_t[list(model)])
        acc = accuracy_score(y_in_t, predicted_y)
        pre = precision_score(y_in_t, predicted_y)
        rec = recall_score(y_in_t, predicted_y)
        if pre > 0.22:
            model_dict[i] = (round(acc,4), round(pre,4), round(rec,4))
            t_results.append((model, (acc, pre, rec)))
        i+=1
    sorted_dict = {k: v for k, v in sorted(model_dict.items(), key=lambda item: item[1][1])}
    return sorted_dict, t_results
```

2.3 Input models to validate. Function returns the validation performance scores of trained models.



```
Accuracy:0.948698508971443
Precision:1.0
Recall:0.004901960784313725
Confusion Matrix
[[   1  203]
 [   0 3753]]
 Coefficients:
[' kw_avg_avg', ' num_hrefs', ' title_sentiment_polarity', ' rate_positive_words', ' num_imgs', ' average_token_length', ' avg_positive_polarity', '
[[ 0.28176516  0.10442487  0.08694787 -0.12547107  0.08047098 -0.10336489
   0.16397063 -0.25489543]]
Intercept:[-3.10005423]
```

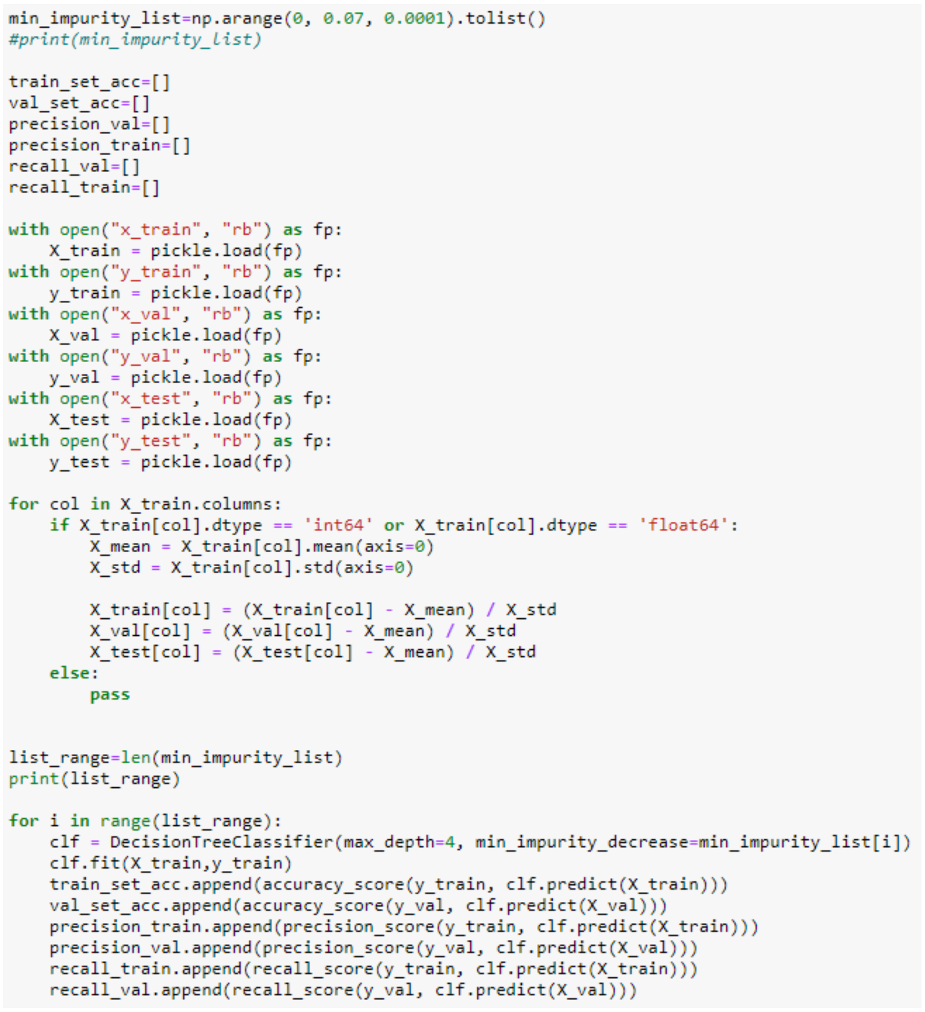
Results for the final LogReg model in test data.

3. Decision Tree

3.1 Max Depth Plotter

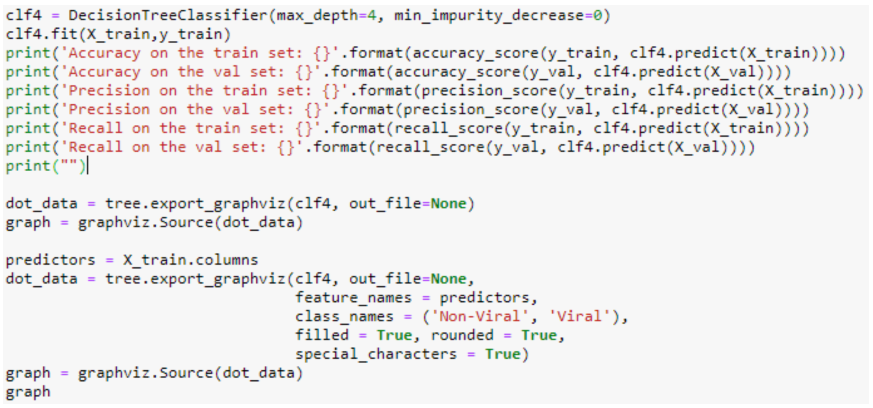
```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.tree import DecisionTreeClassifier
import sklearn.tree as tree
import graphviz
import pickle
import numpy as np
import matplotlib.pyplot as plt

train_set_acc=[]
val_set_acc=[]
precision_val=[]
precision_train=[]
recall_val=[]
recall_train=[]

with open("x_train", "rb") as fp:
    X_train = pickle.load(fp)
with open("y_train", "rb") as fp:
    y_train = pickle.load(fp)
with open("x_val", "rb") as fp:
    X_val = pickle.load(fp)
with open("y_val", "rb") as fp:
    y_val = pickle.load(fp)
with open("x_test", "rb") as fp:
    X_test = pickle.load(fp)
with open("y_test", "rb") as fp:
    y_test = pickle.load(fp)

for col in X_train.columns:
    if X_train[col].dtype == 'int64' or X_train[col].dtype == 'float64':
        X_mean = X_train[col].mean(axis=0)
        X_std = X_train[col].std(axis=0)

        X_train[col] = (X_train[col] - X_mean) / X_std
        X_val[col] = (X_val[col] - X_mean) / X_std
        X_test[col] = (X_test[col] - X_mean) / X_std
    else:
        pass

max_depth_list=list(range(2,16))

for i in max_depth_list:
    clf = DecisionTreeClassifier(max_depth=i, min_impurity_decrease=0)
    clf.fit(X_train,y_train)
    train_set_acc.append(accuracy_score(y_train, clf.predict(X_train)))
    val_set_acc.append(accuracy_score(y_val, clf.predict(X_val)))
    precision_train.append(precision_score(y_train, clf.predict(X_train)))
    precision_val.append(precision_score(y_val, clf.predict(X_val)))
    recall_train.append(recall_score(y_train, clf.predict(X_train)))
    recall_val.append(recall_score(y_val, clf.predict(X_val)))
```

3.2 Decision Tree Min Impurity Decrease Plotter (same imports as Max Depth plotter)

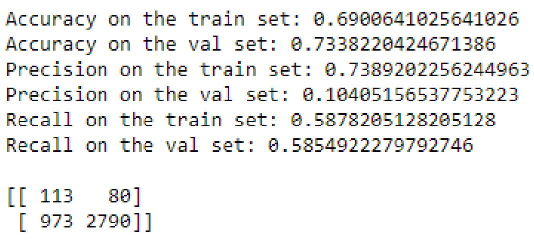


```
min_impurity_list=np.arange(0, 0.07, 0.0001).tolist()
#print(min_impurity_list)

train_set_acc=[]
val_set_acc=[]
precision_val=[]
precision_train=[]
recall_val=[]
recall_train=[]

with open("x_train", "rb") as fp:
    X_train = pickle.load(fp)
with open("y_train", "rb") as fp:
    y_train = pickle.load(fp)
with open("x_val", "rb") as fp:
    X_val = pickle.load(fp)
with open("y_val", "rb") as fp:
    y_val = pickle.load(fp)
with open("x_test", "rb") as fp:
    X_test = pickle.load(fp)
with open("y_test", "rb") as fp:
    y_test = pickle.load(fp)

for col in X_train.columns:
    if X_train[col].dtype == 'int64' or X_train[col].dtype == 'float64':
        X_mean = X_train[col].mean(axis=0)
        X_std = X_train[col].std(axis=0)

        X_train[col] = (X_train[col] - X_mean) / X_std
        X_val[col] = (X_val[col] - X_mean) / X_std
        X_test[col] = (X_test[col] - X_mean) / X_std
    else:
        pass


list_range=len(min_impurity_list)
print(list_range)

for i in range(list_range):
    clf = DecisionTreeClassifier(max_depth=4, min_impurity_decrease=min_impurity_list[i])
    clf.fit(X_train,y_train)
    train_set_acc.append(accuracy_score(y_train, clf.predict(X_train)))
    val_set_acc.append(accuracy_score(y_val, clf.predict(X_val)))
    precision_train.append(precision_score(y_train, clf.predict(X_train)))
    precision_val.append(precision_score(y_val, clf.predict(X_val)))
    recall_train.append(recall_score(y_train, clf.predict(X_train)))
    recall_val.append(recall_score(y_val, clf.predict(X_val)))
```

3.3 Final Tree with Visualisation – data standardized prior



```
clf4 = DecisionTreeClassifier(max_depth=4, min_impurity_decrease=0)
clf4.fit(X_train,y_train)
print('Accuracy on the train set: {}'.format(accuracy_score(y_train, clf4.predict(X_train))))
print('Accuracy on the val set: {}'.format(accuracy_score(y_val, clf4.predict(X_val))))
print('Precision on the train set: {}'.format(precision_score(y_train, clf4.predict(X_train))))
print('Precision on the val set: {}'.format(precision_score(y_val, clf4.predict(X_val))))
print('Recall on the train set: {}'.format(recall_score(y_train, clf4.predict(X_train))))
print('Recall on the val set: {}'.format(recall_score(y_val, clf4.predict(X_val))))
print("")

dot_data = tree.export_graphviz(clf4, out_file=None)
graph = graphviz.Source(dot_data)

predictors = X_train.columns
dot_data = tree.export_graphviz(clf4, out_file=None,
                                feature_names = predictors,
                                class_names = ('Non-Viral', 'Viral'),
                                filled = True, rounded = True,
                                special_characters = True)
graph = graphviz.Source(dot_data)
graph
```



```
Accuracy on the train set: 0.6900641025641026
Accuracy on the val set: 0.7338220424671386
Precision on the train set: 0.7389202256244963
Precision on the val set: 0.10405156537753223
Recall on the train set: 0.5878205128205128
Recall on the val set: 0.5854922279792746

[[ 113   80]
 [ 973 2790]]
```

4. Random Forest – data standardized prior

```
#Random Forest
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(max_depth=4, min_samples_split=4)
model = model.fit(X_train, y_train)
ypred = model.predict(X_val)
acc = accuracy_score(y_val, ypred)
prec = precision_score(y_val, ypred)
rec = recall_score(y_val, ypred)
print('Precision:{}'.format(prec))
print('Recall: {}'.format(rec))
print('Accuracy: {}'.format(acc))
print("")

estimator = model.estimators_[5]
dot_data = tree.export_graphviz(estimator, out_file=None)
graph = graphviz.Source(dot_data)
predictors = X_train.columns
dot_data = tree.export_graphviz(estimator, out_file=None,
                                feature_names = predictors,
                                class_names = ('Negative', 'Positive'),
                                filled = True, rounded = True,
                                special_characters = True)
graph = graphviz.Source(dot_data)
graph
```

```
Precision:0.1024653312788906
Recall: 0.689119170984456
Accuracy: 0.6903437815975733

[[ 133   60]
 [1165 2598]]
```

5. SVM – data standardized prior

```
from sklearn.svm import SVC
model2 = SVC(C=1, kernel='poly', gamma='auto', degree=5)
model2 = model2.fit(X_train, y_train)
ypred2 = model2.predict(X_val)
acc2 = accuracy_score(y_val, ypred2)
prec2 = precision_score(y_val, ypred2)
rec2 = recall_score(y_val, ypred2)
print('Precision:{}'.format(prec2))
print('Recall: {}'.format(rec2))
print('Accuracy: {}'.format(acc2))

from sklearn.metrics import confusion_matrix
con_mat = confusion_matrix(y_val, model2.predict(X_val), labels=[1, 0])
print(con_mat)
```

```
Precision:0.1130820399113082
Recall: 0.26424870466321243
Accuracy: 0.8629929221435794
[[  51  142]
 [ 400 3363]]
```