STA314 Final Project

Detecting spam YouTube comments

Kaggle group name: Defy-gravity-MLE

Kaggle test score: 93.66%

Kaggle final ranking: 27

Quercus group name: 112

Group Members:

1. Rafsaan Sanvir
    a. Student ID: 1007447522
    b. Email: rafsaan.sanvir@mail.utoronto.ca
2. Shiva Chandrachary
    a. Student ID: 1009674461
    b. Email: shiva.chandrachary@mail.utoronto.ca
3. William Kwan
    a. Student ID: 1007158898
    b. Email: willl.kwan@mail.utoronto.ca
4. Xiangdi Wang
    a. Student ID: 1008472261
    b. Email: caitlyn.wang@mail.utoronto.ca

Github:

Main page: https://github.com/schandrachary/STA314

Classifier code:
https://github.com/schandrachary/STA314/blob/master/Scripts/classifier.py

Problem Statement

Youtube is a significant platform for both entertainment and information, with 2.7 billion monthly users interacting with this platform. However, its popularity has attracted spammers and bots who post irrelevant or promotional comments that disrupt the user experience of the platform. This can decrease the level of user engagement on the platform, and lower user engagement means lower revenue for the company as a whole. Manual removal of comments is both impractical and time-intensive, which calls for the need for automated solutions.

Classification algorithms are scalable and can efficiently detect spam comments without user intervention. This project leverages natural language processing techniques and machine learning to identify spam comments with a high degree of accuracy, which can then be removed to improve the quality of content and user engagement on youtube.

Statistical Analyses

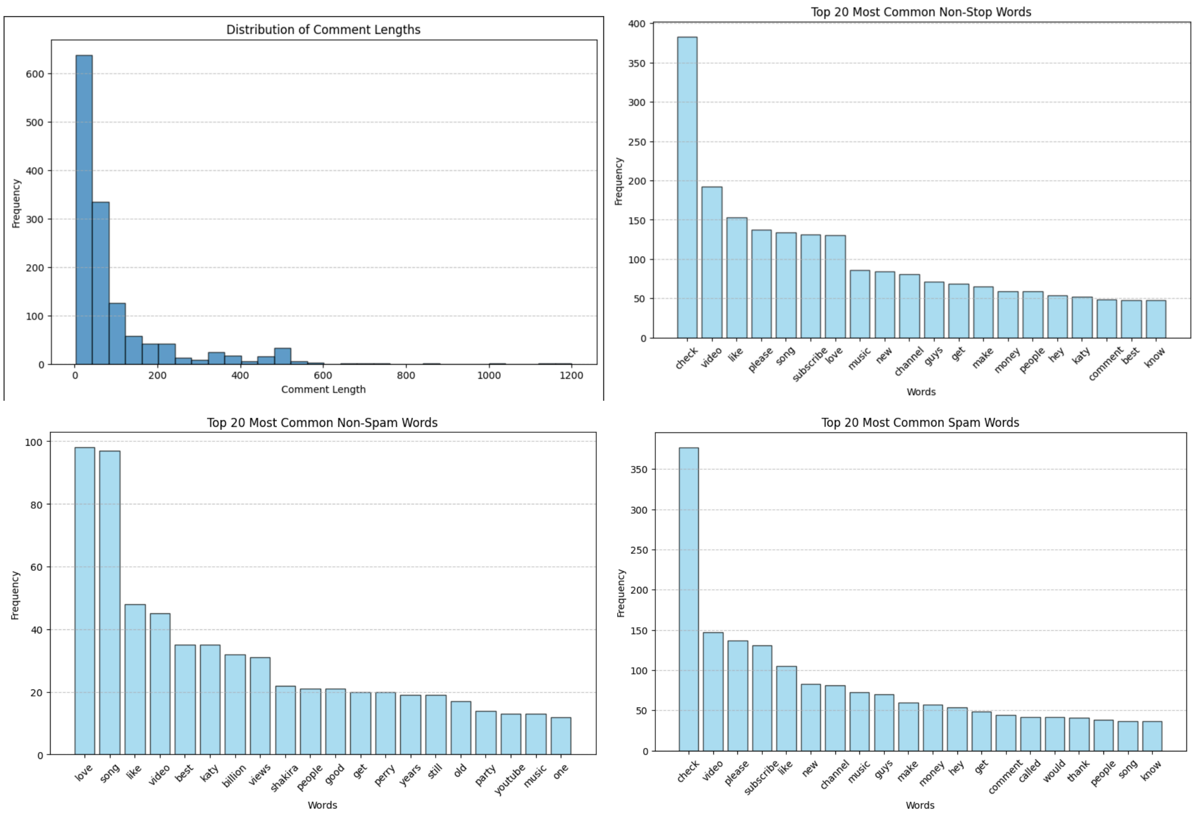


Figure 1a, 1b,c,d. Distribution of Comment Length and most common words between classes

Comments	Total Comments: 1369 Length: 3 - 1200 Mean Length: 96 characters
Words in comments	Mean: 15.82 words Number of words: 1 - 172
Class distribution	51.86% spam, 48.14% not spam.
Most common non-stop words for each class (Fig 1c, d)	Spam: check, video, please, subscribe, like Not spam: love, song, like, video, best

Average similarity for each class	Spam: 64.53% Not Spam: 7.49%
Most correlated words to class:	\`check\`: 0.5322 \`subscribe\`:0.3557 \`please\`: 0.3071

Table 1: Dataset statistics

In the dataset, we have the features COMMENT_ID, AUTHOR, DATE, CONTENT, VIDEO_NAME, and CLASS. We extract the relevant features CONTENT and CLASS.

First, we see that the dataset is very skewed (Fig 1a). More than 50% of comments are 48 characters or smaller, and 75% of comments are under 99 characters. Next, the proportion of data between classes is fairly balanced, so there is no extreme class imbalance. We see that common non-stop words are different for each class, for example, in spam comments, the most common words are ‘check’, ‘video’, and ‘subscribe’, with 377, 147, and 137 occurrences, respectively. While in non spam comments, the most common words are ‘love’, ‘song’, and ‘like’, with 98, 97 and 48 occurrences respectively. Finally, if we were to train a Word2Vec model and calculate the cosine similarity between words in spam and non spam comments, we see that spam comments are highly similar to each other (64%), while non-spam comments are rarely similar (7%).

Null values and data types: Some important data cleaning procedures include checking for null values in a dataset, and ensuring that the datatype of each row is correct which can impact our classification task. Rows with null values either have to be filled in or deleted and after checking our data, there were 0 null values present in the “comments” and “class” column. A function was written to ensure that the comments are of type string and the class has all integer values.

Data Cleaning using stop-words

Stop words are typically the most frequently used words in a language and are generally considered to have little semantic meaning in the context of text analysis. These words are often removed from text data during preprocessing because they can introduce noise without contributing significant information about the content or meaning of the text. Examples in English include "the", "is", "in", "at", "which", "and", etc. Many stop words are function words, which serve a grammatical purpose rather than conveying meaning. They help in structuring sentences but do not provide much insight into the topic or sentiment of the text.

By removing the stop words, the size of the vocabulary is reduced. We try to focus on words that carry important information, such as nouns, verbs, adjectives, and adverbs. By reducing the feature space of the model, we decrease the complexity and improve their performance. See below for a list of stop words.

Stop words used in our data cleaning process (see appendix for a full list):

```
 ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're",
"you've", "you'll", "you'd", 'your', 'yours', 'yourself', etc...]
```

The process involved in data cleaning for our model:

1. First, we converted all the words into lower-case letters.
2. We experimented with removal of links, for example words starting with "http", "www", "https". We decided to keep these words as these features are deemed to be significant in model prediction.
3. We removed all non-ASCII characters.
4. We removed any characters that are not an alphabet, a digit, or a whitespace.

5. We removed whitespace characters such as spaces, tabs, and new lines.
6. We removed any digit, equivalent to `[0-9]`.
7. We removed any trailing whitespace characters from a string either at the beginning or at the end of the sentences.

Feature Selection

To convert our text into a format suitable for machine learning models, we considered three vectorization options and chose TF-IDF for most models. While Bag of Words focuses solely on the raw frequency of terms within a document, TF-IDF enhances this by incorporating the inverse document frequency, providing a more nuanced representation that highlights terms significant to a document while down-weighting common words across the corpus. The 3 vectorization techniques are explained below:

Vectorizing features with the TF-IDF vectorizer

The Tf-IDF vectorizer takes a sentence as an input and gives us a numeric output vector which can then be fed into the different machine learning models we implemented. The calculation has 2 components:

$$TF = \frac{\#\ of\ times\ a\ word\ appears\ in\ the\ sentence}{\#\ of\ words\ in\ the\ sentence}\ ;\ IDF = log(\frac{Total\ \#\ of\ sentences}{\#\ of\ sentences\ the\ word\ appears\ in})$$

Term frequency (TF) measures the occurrence of a term/feature in the sentence relative to the total number of terms in the sentence, and inverse domain frequency score downweights the terms that appear frequently across multiple sentences, and gives more weight to rarer features/words.

Multiplying the 2 scores above gives us TF-IDF score of each feature, $TF\ IDF\ score\ = TF\ *\ IDF$, which is stored in the output vector. TF-IDF provides a meaningful representation of the text, emphasising words that are specific to certain comments while downplaying generic ones like "the" or "and."

Vectorizing features with Bag of Words

The Bag-of-Words (BoW) model is a classic technique in natural language processing (NLP) for representing textual data as numerical vectors.

In this dataset, each unique word works as a feature, and all of the words form a vocabulary. The vector representation of a comment corresponds to the frequency of each word. Some words like 'check' appear more frequently in spam comments, and the BoW model can directly reflect the frequency of the words, which helps to identify patterns in spam comments. In addition, it enables machine learning algorithms to train the model.

Vectorizing features with Word-to-vec vectorizer

In the case of vectorizing comments with Word2Vec, it requires a large corpus to learn word embeddings, and as such, we use pre-trained Google News vectors [9], where each word in the comments is mapped to a 300-dim vector from a model trained on 100 billion words using continuous bag of words. For each word in the comment, if it is found in the pre-trained vocabulary, it is assigned its corresponding vector.

In the end, we chose TF-IDF vectorizer due to its suitability for short texts, in which we can extract features within the domain of our dataset rather than needing to pre-train from a larger dataset. Moreover, Word2Vec is designed to capture patterns within text (n-grams), in which Youtube comments might not be complex or diverse enough to be captured effectively. Hence, it is more suitable for our smaller dataset, in which assigning words based on their frequency is more important than capturing their meaning in Word2Vec.

Features

The process described above on data-cleaning, and then converting words into tokens can be performed with a library called `TfidfVectorizer`, with a single line:

```
feature_extraction = TfidfVectorizer(min_df = 2, stop_words =
'english', lowercase = True)
```

Here, `min_df = 2` removes words that appear in less than 2 sentences. `stop_words = english` uses the same process for data cleaning as described in the data cleaning section above. This function returns a feature model that can then be used to get tokens from our training dataset. This process results in a total of 884 features for our training dataset.

Max_df argument for the tf-idf vectorizer filters out terms that appear in more than the specified proportion of documents, and max_df = 1 is used by our vectorizer by default. Max_df = 0.7 was the best parameter found after performing gridsearch, which translates to "do not use words as features that appear in more than 70% of sentences". However applying this parameter to our vectorizer did not reduce the amount of features already present after applying min_df = 2.

These parameters were chosen to give us the most simplistic model yet achieving highest performance.

Model Training

GridSearchCV enhances model performance by systematically searching through a set of hyperparameters to find the best combination that gives us optimal model performance, evaluated via accuracy score. To address the issue of having only a training dataset we used 5-fold Cross-Validation (CV), which ensured robust model evaluation and optimization even with a limited training dataset.

First, we randomly split the available train dataset into two parts: a training set and a validation set. The training set (80% of the data) is used for model training, while the validation set (20% of the data) is used for evaluating the fitted model.

Next, the CV approach involves randomly dividing the training set into 5 equal-sized folds. The model is trained and evaluated multiple times, each time using one fold as the validation set and using the remaining folds for training.

This approach ensures that the model is trained on the majority of the data, while maintaining the stability of model training. It also helps to identify the best hyperparameters for the model. Furthermore, the separate test subset can be used to evaluate the fitted model performance.

Metrics

We picked accuracy score as a metric to measure model performance since it is easy to implement and interpret. It measures the proportion of correct classifications made by our model out of all the predictions, mathematically described as: $Accuracy = \frac{\# \, of \, correct \, predictions}{Total \, \# \, of \, predictions}$

While accuracy works well for balanced datasets, it may be misleading in cases of class imbalance. For example, if spam comments are rare, the model might achieve high accuracy by simply predicting "non-spam" most of the time.

And we also picked F1 Score and AUC as metrics to evaluate the performance of the model.

$F1\ score\ =\ \frac{2\,(\#\ of\ TP)}{2\,(\#\ of\ TP) + (\#\ of\ FP) + (\#\ of\ \ FN)}$ ranges from 0 to 1, where TP (True Positive) is correctly predicted positive instances, FP (False Positive) is incorrectly predicted positive instances and FN (False Negative) is incorrectly predicted negative instances.

AUC is the area under the ROC curve, which plots the True Positive Rate (TPR) against the False Positive Rate (FPR). And it is ranging from 0 to 1.

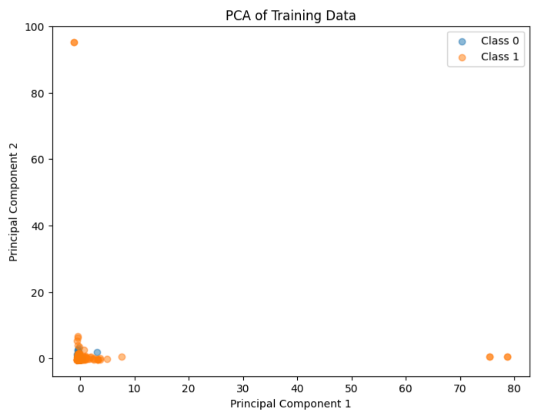
F1 Score combines and balances both precision and recall metrics, and is considered a valuable metric for our model selection. Both a higher F1 Score and AUC means better model performance. However, if the two metrics are closed to 1, the model may be overfitted.

Results and Conclusion

For this classification problem, both Binary Logistic Regression (LR) and Support Vector Machine (SVM) are valid models to train and predict the datasets on. Both models try to draw a hyperplane that separates the two classes; however Logistic Regression is a linear model, and SVM allows non-linear relationships between features and the binary outcome. (add decision tree/neural networks). Decision trees might be prone to overfitting (high train accuracy) especially since there are sparse features with Tf-Idf, while Convolutional Neural Networks (CNN) must require dense embeddings using a pretrained Word2Vec model from a large dataset, and the dataset given to us in Kaggle doesn't satisfy this requirement.

Since LR and SVM were very close in terms of performance, we assess these two models more closely below. Even though Logistic Regression can handle non-linearity, SVM is more flexible when it comes to non-linear features. The final model we chose was SVM and we try to explain below why we made that choice.

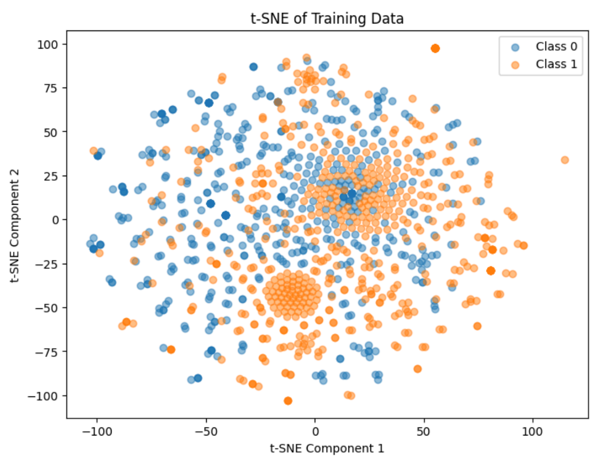
First, let's assess the dataset for linearity, non-linearity, clusters, outliers, etc. PCA plot below shows how the data is distributed along the first two principal components. These capture the direction of maximum variance in the data.

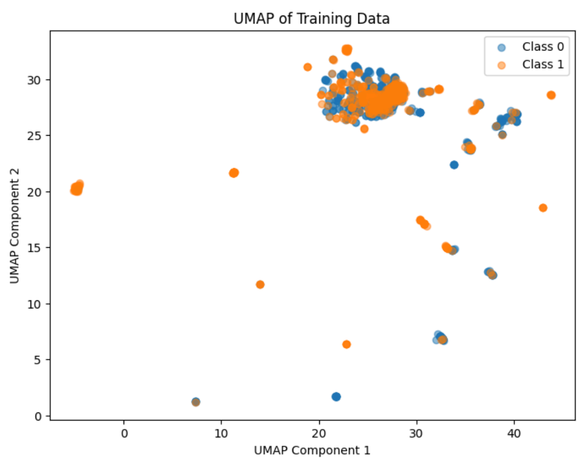


Since we see one cluster near the origin we see a common pattern between spam and not spam. But looking closely, there is a significant overlap between the two classes in that cluster. It can be very challenging for a linear model like Logistic Regression to classify these points in this scenario.

There are also points further away from the cluster, these could be outliers or potentially unique samples.

Since PCA performs a linear transformation from higher dimension to lower dimension, it may not capture complex, non-linear relationships. Since we see data in the cluster being not well separated, we will consider t-SNE and UMAP for capturing the non-linear patterns. See these plots below.





t-distributed Stochastic Neighbour Embedding:

The above plot shows how the data is distributed in the first two significant components of t-SNE. We see one cluster where the data is very well separated. This might indicate why we see a good prediction model from the Logistic Regression model. However, if you look closely in the second cluster, you can see the data between two classes are not very well separated. t-SNE is good at preserving local structures, so points that are close together in this plot are likely similar in the high-dimensional space.

There are also many points that are scattered further away from these two main clusters. These could either be outliers or unique samples that must be treated with care.

Next, we can also take a look at UMAP, which preserves both local and global structures of the data better than t-SNE.

UMAP: There appears to be one large cluster and many tiny clusters distributed throughout the plot. The large cluster also seems to have a lot of overlapping between two classes, which indicates difficulty in class separation. This is also something we saw in the t-SNE plot. Once again, we see quite a few isolated points, which either suggest outliers, or unique samples that need to be handled with care.

One thing we can quickly notice are the clusters that form spam labels. Notice one near (3,20), and one around (30,17), and a few other similar clusters. These might suggest a stronger pattern in the data. As discussed in the Statistical Analyses section above, these might correspond to particular words such as "check" , "video", "please" and, "subscribe" as this indicates a strong pattern.

Based on the above discussion, we might as well consider looking at the outlier observations. For this, let’s look at the Distance-Based method called $Z - Score$. It calculates the Z-score for each feature and removes samples with scores above a certain threshold, for example, samples above 3 standard deviations.

```
Original data shape: (1095, 884)     Cleaned data shape: (25, 884)
```

Removing all samples above 3 standard deviations took our sample space from 1095 to just 25 samples. This removes data points that are identified as outliers based on the threshold from the mean value. We need to understand those samples for their unique features before removing them completely.

Below is the SVM model performance from the Z-score based outlier removal. LR also gave a very similar result (not mentioned below).

```
Accuracy of training data using Logistic Regression: 0.96
Accuracy of test data using Logistic Regression: 0.5437956204379562
```

Looks like Z-Score removed some important observations that are used for classification. After all, those samples above 3 std may not be outliers.

We will also look at an alternative to Z-Score for outlier detection, called Isolation Forest, which is a tree-based model that isolates outliers by randomly partitioning the data.

```
iso_forest = IsolationForest(contamination=0.05, random_state=42)
Original data shape: (1095, 884)          Cleaned data shape: (1040, 884)
```

Points isolated quickly are likely outliers. A `contamination` value of 0.05 specifies the proportion of the dataset that is expected to be outliers. `random_state` is used like a seed value for reproducing the same result and a value of 42 is common in practice. This method reduces our sample space from 1095 to 821. In contradiction to Z-score, this says those cluster pockets we saw in the plots above, are not really outliers, but have some unique properties in them that are helpful for classification.

```
Accuracy of training data using SVM: 0.9971153846153846
Accuracy of test data using SVM: 0.9416058394160584
```

Using gridsearch cross-validation method to find the best parameters for SVM model gives the following result. `Best parameters:`

```
{'C': 1, 'degree': 2, 'gamma': 'scale', 'kernel': 'linear'}
Accuracy on training data: 0.9807692307692307
Accuracy on test data: 0.9562043795620438
```

Compare it to the previous run of gridsearch CV that overfitted the training samples:

```
Best parameters:{'C': 8, 'degree': 0, 'gamma': 0.5, 'kernel': 'rbf'}
Accuracy on training data: 0.9971153846153846
Accuracy on test data: 0.9416058394160584
```

We purposely chose some parameters that weren't overfitting the model. Moreover, from our discussion above, we want to look for a model that can fit a polynomial of degree > 1 since we see non-linearity in the data. A comparison with LR is given below. Although LR gave a better result in the training dataset, it performed poorly in the test data on kaggle.

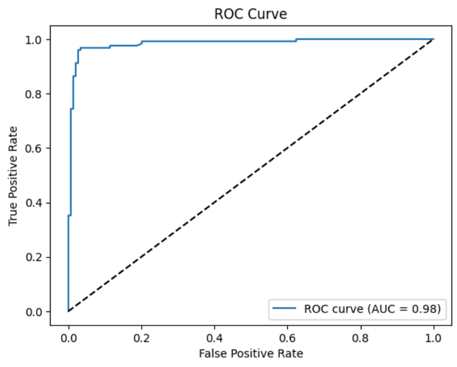
```
Accuracy of training data using Logistic Regression: 0.9617307692307692
Accuracy of test data using Logistic Regression: 0.9398540145985401
```

SVM Model (Our final model):

The final model we chose is SVM with parameters: `C=1,gamma=0.5.`
A low value for C, the regularization parameter, such as 1, means that the model will tolerate some errors in the training data in exchange for a simpler model that might generalize better. A gamma value of 0.5 indicates that the model is creating a complex decision boundary, likely because of non-linearity in the dataset. The default kernel `rbf` is used in this model which maps the input features into a higher-dimensional space where a linear decision boundary can separate the data that is not linearly separable in the original feature space.

This model gave us 94.7% accuracy on the test dataset on Kaggle.

```
False Positives: 3, False Negatives: 13, True Positives: 112
F1 Score: 93.33%, AUC: 0.9842
```



The ROC shows an excellent ability to classify spam and non-spam comments. The curve hugs the top-left corner of the plot, indicating a high sensitivity and low FPR, and stays significantly above the diagonal baseline.

Other models considered:

Decision Tree: Using the TF-IDF features, we also used a naive Decision Tree classifier. This gave us a 93.8% test accuracy on the test dataset. Applying GradientBoosting improved the test accuracy to 94.1%.

Neural Networks: Using the input dimension of 300 from Word2Vec, a CNN model applying 3 parallel 1D convolutional layers with kernel sizes 3, 4, and 5, each using 100 filters was trained. After each convolution, ReLU activation and max-pooling are applied. The output is concatenated into a 300-dim feature vector followed by a dropout layer (0.5 probability) to prevent overfitting. A fully connected layer then processes the features, with a sigmoid activation to output probabilities. This model gives us a 92% test accuracy on the test dataset on Kaggle.

Discussion

From the Result and Conclusion above, the final model shows its ability to identify whether a YouTube comment is spam or not, allowing the YouTube platform to filter and hide the spam comments to protect users and foster a better and healthier commenting environment.

However, during the model training and selection process, we found several limitations of our model. One major issue is that the model overly relied on specific words that frequently appear in spam comments. For example, if a comment contains the word ’check’, it is likely to be classified as spam regardless of the rest of the content, which leads to misclassifications and blocking reasonable comments on the platform. Additionally, the model performs poorly when dealing with misspelling situations. For instance, spam comments that use misspelled words such as ‘ch3ck’ or ‘checkkk’ to replace ‘check’, often be classified as not spam. Another limitation is that the model cannot identify spam comments well when processing complex sentences. If a spam comment deliberately avoids using frequently flagged words or uses rare syntax or semantics, the model may misclassify it. Also, our group did not consider ensemble tree techniques such as bagging, boosting and random forests which could potentially improve the mode performance.

We believe that with more diverse data, for example sources other than music videos, can help the model learn and detect a wider range of spam patterns. We note that spam comments in music videos are different from those in, for example, vlogs. More generally, machine learning techniques such as those presented in this report can also be applicable to other social media platforms, such as emails, Twitter and Instagram. We conclude that a future direction would be to broaden the data and applications of the spam detection algorithm. These methods can enhance the model’s ability to detect spam comments and reduce the probability of misclassification.

References:

1. A Beginner’s Guide to Tokens, Vectors, and Embeddings in NLP | by Sascha Metzger | Medium
2. TF-IDF Vectorizer Explained. The Term Frequency-Inverse Document… | by Anurag | Medium
3. In Depth: Parameter tuning for SVC | by Mohtadi Ben Fraj | All things AI | Medium
4. How to detect outliers with z-score - Machine Learning Plus
5. Anomaly Detection with Isolation Forest and Kernel Density Estimation - MachineLearningMastery.com
6. How to Build a Logistic Regression Model: A Spam-filter Tutorial - DEV Community
7. Powerful Comparison: TF-IDF vs Bag of Words
8. Principal Component Analysis for Visualization - MachineLearningMastery.com
9. CNN for Sentence Classification - Yoon Kim
10. CNN Text Classification using Pytorch

Code for SVM and Logistic Regression Model comparison

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn import svm
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, confusion_matrix


# Read the CSV file, convert it to a dataframe and return the dataframe
# Also returns a success parameter. If the passed fileName is invalid,
# return failure status.
def readData(fileName):
    success = False
    dataFrame = pd.DataFrame()

    if fileName.lower().endswith('.csv'):
        try:
            dataFrame = pd.read_csv(fileName)
            success = True
            return dataFrame, success
        except FileNotFoundError:
            print(f"File {fileName} not found. Please check the file
path and try again.")
        except Exception as e:
            print(f"An error occurred while reading the file: {e}")
    else:
        return dataFrame, success
```

```
# Use readData method to read a given training .csv file.
# Load the "content" and "class" column of the dataframe and return
# X as features and Y as responses
def loadTrainingData(fileName):
    # Read the data and store it as a data frame
    dataFrame, _ = readData(fileName)
    dataFrame.drop(["AUTHOR", "DATE", "CONTENT", "VIDEO_NAME"], axis =
1)
    X = dataFrame["CONTENT"]
    Y = dataFrame["CLASS"]

    return X, Y

# Use readData method to read a given test .csv file.
# Load the "content" column of the dataframe and return
# X as features
def loadTestData(fileName):
    # Read the data and store it as a data frame
    dataFrame, _ = readData(fileName)
    dataFrame.drop(["AUTHOR", "DATE", "CONTENT", "VIDEO_NAME"], axis =
1)
    X = dataFrame["CONTENT"]

    return X, dataFrame

# Fit the model to the a given data set, and return the fitted model
def fitModel(model, X, Y):
    model.fit(X, Y)
    return model

# Predict the observation for a given model and its feature, X
```

```
def predictFeatures(model, X):
    Y = model.predict(X)
    return Y

# Compute accuracy between a predicted observation and true observation
# Accuracy score = # of correct predictions / Total # of predictions
# Accurary = 1 - TRAINING ERROR RATE
# Y1: True observation
# Y2: Predicted observation
# testType: Type of the test data passed, for example "training" or
# "test" data
# modelName: Name of the model used, for example "SVM" or "Logistic
Regression"
def computeAccuracy(Y1, Y2, testType, modelName):
    score = accuracy_score(Y1, Y2)
    print(f"Accuracy of {testType} using {modelName}: {score}")
    print(f"Classification report of {testType} with {modelName}
model:\n {classification_report(Y1, Y2)} ")



#######################################################
# Load training data set and vectorize it
#######################################################

# Read the dataset and create a training set with X and Y
Xtrain, Ytrain = loadTrainingData("../Dataset/train.csv")

# Split the training dataset with 80-20 split
Xtrain_train, Xtrain_test, Ytrain_train, Ytrain_test = \
train_test_split(Xtrain, Ytrain, test_size = 0.2, random_state = 3)
```

```
# Change the min_df from 1 to 2, i.e, if a word appears in less than 2
sentences, drop it.

feature_extraction = TfidfVectorizer(min_df = 2, stop_words =
'english', lowercase = True)

Xtrain_train_features = feature_extraction.fit_transform(Xtrain_train)

Xtrain_test_features = feature_extraction.transform(Xtrain_test)


#let's make sure the labels for Y are in int form e.g 0, 1 and not any
other like "0", "1"

Ytrain_train = Ytrain_train.astype('int')

Ytrain_test = Ytrain_test.astype('int')


#######################################################

# Load test data set and vectorize it

#######################################################


comments_test, df_test = loadTestData("../Dataset/test.csv")


# Use TF-IDF vectorizer to vectorize test data and extract features

Xtest_test_features = feature_extraction.transform(comments_test)


#######################################################

# Begin SVM Model

#######################################################


# Create a parameter grid for selecting the best model

# param_grid = {'C': [0.1, 1, 5, 8, 10, 100],

#               'gamma': [1, 0.5, 0.3, 0.2, 0.1, 0.09, 0.01],

#               'degree':[0, 1, 2, 3],

#               'kernel': ['rbf', 'linear']}


# Fit the training features to the SVM model. Use the vectorized data
# from TF-IDF vectorizer

# svmModel = GridSearchCV(svm.SVC(), param_grid, refit = True, verbose
= 3)


# Model parameters chosen from cross-validation.

svmModel = svm.SVC(C=1, degree=0, gamma=0.5)

svmModel = fitModel(svmModel, Xtrain_train_features, Ytrain_train)


# Predict traning and test dataset

predictionTrain_train = predictFeatures(svmModel,
Xtrain_train_features)

predictionTrain_test = predictFeatures(svmModel, Xtrain_test_features)

computeAccuracy(Ytrain_train, predictionTrain_train, "training data",
"SVM")

computeAccuracy(Ytrain_test, predictionTrain_test, "test data", "SVM")



###### Test SVM Model ########

# Perform prediction on test data set

df_test["CLASS"] = predictFeatures(svmModel, Xtest_test_features)


#Store classified result in a .csv file

df_test.to_csv("../Dataset/output/svmClass.csv", index=False)


#######################################################

# End SVM Model

#######################################################


#######################################################

# Begin Logistic Regression Model

#######################################################
```

```
# Create an instance of a logistic regression model

logisticRegModel = fitModel(LogisticRegression(C=4.281332398719396,
solver='saga'), Xtrain_train_features, Ytrain_train)


# Predict traning and test dataset

predictionTrain_train = predictFeatures(logisticRegModel,
Xtrain_train_features)

predictionTrain_test = predictFeatures(logisticRegModel,
Xtrain_test_features)

computeAccuracy(Ytrain_train, predictionTrain_train, "training data",
"Logistic Regression")

computeAccuracy(Ytrain_test, predictionTrain_test, "test data",
"Logistic Regression")


###### Test Logistic Regression Model ########

# Perform prediction on test data set

df_test["CLASS"] = predictFeatures(logisticRegModel,
Xtest_test_features)


#Store classified result in a .csv file

df_test.to_csv("../Dataset/output/logisticRegressionClass.csv",
index=False)


#######################################################

# End Logistic Regression Model

#######################################################
```

Stop words in english: ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before',

```
'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off',
'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when',
'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most',
'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than',
'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should',
"should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't",
'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't",
'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't",
'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't",
'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

Code for statistical analysis

```
###### Library imports ######

import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix,
roc_auc_score, roc_curve

import matplotlib.pyplot as plt

import os

import re

from tqdm import tqdm

import seaborn as sns

import nltk

from collections import Counter

from wordcloud import WordCloud

from itertools import combinations

from sklearn.ensemble import AdaBoostClassifier,
GradientBoostingClassifier

from sklearn.tree import DecisionTreeClassifier

from gensim.models import Word2Vec

import torch

from torch.utils.data import TensorDataset, DataLoader
```

```
import torch.nn as nn

import torch.nn.functional as F


# Download all nltk resources

nltk.download("all")
```

"""Add a 'Content_Length' column and visualize the distribution of comment lengths."""

```
# Add a column for the lengths of the comments

df['Content_Length'] = df['CONTENT'].apply(len)


# Plot the distribution of the lengths

plt.figure(figsize=(10, 6))

plt.hist(df['Content_Length'], bins=30, edgecolor='k', alpha=0.7)

plt.title('Distribution of Comment Lengths')

plt.xlabel('Comment Length')

plt.ylabel('Frequency')

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.show()

print(df['Content_Length'].describe())
```

"""Generate a statistical summary of comment lengths."""

```
review_lengths = df["CONTENT"].dropna().apply(len)

summary = {

    "Total Reviews": len(review_lengths),

    "Minimum Length": review_lengths.min(),

    "Maximum Length": review_lengths.max(),

    "Average Length": review_lengths.mean(),

    "Median Length": review_lengths.median(),

    "Standard Deviation": review_lengths.std(),

    "Length Quartiles": review_lengths.quantile([0.25, 0.5, 
0.75]).to_dict()
}
# Display the summary
for key, value in summary.items():
    print(f"{key}: {value}")
"""Calculate and visualize the proportions of each class."""
class_proportions = df['CLASS'].value_counts(normalize=True)
# Display the proportions
print("Class Proportions:")
print(class_proportions)
# Plot the proportions
plt.figure(figsize=(8, 8))
class_proportions.plot(
    kind='pie',
    autopct='%1.1f%%',
    startangle=90,
    colors=['skyblue', 'orange'],
    labels=class_proportions.index,
    explode=[0.05, 0]
)
plt.title('Proportion of Each Class')
plt.ylabel('')
plt.show()
"""Identify and display the most common words in the entire dataset."""
# Combine all text in 'CONTENT' into a single string
all_text = " ".join(df["CONTENT"].dropna())
word_counts = Counter(all_text.split())
most_common_words = word_counts.most_common(10)
print("Most common words:")
for word, count in most_common_words:
    print(f"{word}: {count}")
"""Filter out stop words and count the most common non-stop words."""
```

```
# Download stopwords

nltk.download('stopwords')

stop_words = set(stopwords.words('english'))


def get_non_stop_words(content):

    words = content.split()

    non_stop_words = [word.lower() for word in words if word.lower()
not in stop_words and word.isalpha()]

    return non_stop_words

# Extract all non-stop words

all_words = []

df['CONTENT'].apply(lambda content:
all_words.extend(get_non_stop_words(content)))

# Count the most common non-stop words

word_counts = Counter(all_words)

most_common_words = word_counts.most_common(20)

# Display the most common non-stop words

print("Most Common Non-Stop Words:")

for word, count in most_common_words:

    print(f"{word}: {count}")

# Plot the most common non-stop words

plt.figure(figsize=(10, 6))

plt.bar(*zip(*most_common_words), color='skyblue', edgecolor='k',
alpha=0.7)

plt.title('Top 20 Most Common Non-Stop Words')

plt.xlabel('Words')

plt.ylabel('Frequency')

plt.xticks(rotation=45)

plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.show()

"""Train Word2Vec models and calculate average word similarity for each
class."""

def train_word2vec(corpus):
    tokenized = [sentence.split() for sentence in corpus]
    model = Word2Vec(sentences=tokenized, vector_size=100, window=5,
min_count=1, workers=4)
    return model
def calculate_average_similarity(words, model):
    word_vectors = [model.wv[word] for word in words if word in
model.wv]
    if len(word_vectors) < 2:
        return 0
    similarities = cosine_similarity(word_vectors)
    upper_triangle = np.triu_indices(similarities.shape[0], k=1)
    return similarities[upper_triangle].mean()
# Train Word2Vec models for each class
class_0_model = train_word2vec(df[df['CLASS'] == 0]['CONTENT'])
class_1_model = train_word2vec(df[df['CLASS'] == 1]['CONTENT'])
# Calculate average similarities
class_0_words = " ".join(df[df['CLASS'] == 0]['CONTENT']).split()
class_1_words = " ".join(df[df['CLASS'] == 1]['CONTENT']).split()
class_0_similarity = calculate_average_similarity(class_0_words,
class_0_model)
class_1_similarity = calculate_average_similarity(class_1_words,
class_1_model)
print(f"Average Similarity (Class 0): {class_0_similarity:.4f}")
print(f"Average Similarity (Class 1): {class_1_similarity:.4f}")
```

Code for CNN Model creation

```
"""Create Tensor datasets and DataLoaders for PyTorch training and
testing."""
import torch
from torch.utils.data import TensorDataset, DataLoader


# Create Tensor datasets
```

```
train_data = TensorDataset(torch.from_numpy(train_x),
torch.from_numpy(train_y))

valid_data = TensorDataset(torch.from_numpy(val_x),
torch.from_numpy(val_y))

test_data = TensorDataset(torch.from_numpy(test_x),
torch.from_numpy(test_y))


# Dataloaders with batch size

batch_size = 50

train_loader = DataLoader(train_data, shuffle=True,
batch_size=batch_size)

valid_loader = DataLoader(valid_data, shuffle=True,
batch_size=batch_size)

test_loader = DataLoader(test_data, shuffle=True,
batch_size=batch_size)
```

"""Define a CNN model for comment analysis using an embedding layer and convolutional layers."""

```
class CommentCNN(nn.Module):

    def __init__(self, embed_model, vocab_size, output_size,
embedding_dim,

                 num_filters=100, kernel_sizes=[3, 4, 5],
freeze_embeddings=True, drop_prob=0.5):

        super(CommentCNN, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        self.embedding.weight =
nn.Parameter(torch.from_numpy(embed_model.vectors))


        if freeze_embeddings:

            self.embedding.requires_grad = False

        self.convs_1d = nn.ModuleList([

            nn.Conv2d(1, num_filters, (k, embedding_dim), padding=(k -
2, 0))

            for k in kernel_sizes

        ])
        self.fc = nn.Linear(len(kernel_sizes) * num_filters,
output_size)
        self.dropout = nn.Dropout(drop_prob)
        self.sig = nn.Sigmoid()
    def conv_and_pool(self, x, conv):
        x = F.relu(conv(x)).squeeze(3)
        x_max = F.max_pool1d(x, x.size(2)).squeeze(2)
        return x_max

    def forward(self, x):
        embeds = self.embedding(x).unsqueeze(1)
        conv_results = [self.conv_and_pool(embeds, conv) for conv in
self.convs_1d]
        x = torch.cat(conv_results, 1)
        x = self.dropout(x)
        logit = self.fc(x)
        return self.sig(logit)
"""Train the CNN model using PyTorch with a specified number of epochs
and track validation loss."""
lr = 0.001
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

def train(net, train_loader, epochs, print_every=100):
    if train_on_gpu:
        net.cuda()

    counter = 0
    net.train()

    for e in range(epochs):
        for inputs, labels in train_loader:
            counter += 1

            if train_on_gpu:
                inputs, labels = inputs.cuda(), labels.cuda()

            net.zero_grad()
            output = net(inputs)
            loss = criterion(output.squeeze(), labels.float())
            loss.backward()
            optimizer.step()

            if counter % print_every == 0:
                val_losses = []
                net.eval()
                for inputs, labels in valid_loader:
                    if train_on_gpu:
                        inputs, labels = inputs.cuda(), labels.cuda()
                    val_loss = criterion(output.squeeze(), 
labels.float())
                    val_losses.append(val_loss.item())
                net.train()
                print(f"Epoch: {e+1}/{epochs}... Step: {counter}... 
Loss: {loss.item():.6f}... Val Loss: {np.mean(val_losses):.6f}")
epochs = 10
print_every = 100
train(net, train_loader, epochs, print_every=print_every)
```

Code for CNN test accuracy

```
"""Evaluate the model on test data and calculate accuracy."""
test_losses = []
```

```
num_correct = 0

net.eval()

for inputs, labels in test_loader:

    if train_on_gpu:

        inputs, labels = inputs.cuda(), labels.cuda()

    output = net(inputs)

    test_loss = criterion(output.squeeze(), labels.float())

    test_losses.append(test_loss.item())

    pred = torch.round(output.squeeze())

    correct_tensor = pred.eq(labels.float().view_as(pred))

    correct = np.squeeze(correct_tensor.cpu().numpy()) if train_on_gpu
else np.squeeze(correct_tensor.numpy())

    num_correct += np.sum(correct)

print(f"Test loss: {np.mean(test_losses):.3f}")

test_acc = num_correct / len(test_loader.dataset)

print(f"Test accuracy: {test_acc:.3f}")
```

Code for Decision tree with boosting

```
"""Train an AdaBoost classifier with a Decision Tree base model."""

# Base model: DecisionTreeClassifier

base_model = DecisionTreeClassifier(max_depth=1)

boosted_model = AdaBoostClassifier(estimator=base_model,
n_estimators=100, learning_rate=0.1)

boosted_model.fit(X_train_features, Y_train)

"""Train a Gradient Boosting classifier and evaluate accuracy on test
data."""

gb_model = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, max_depth=3)

gb_model.fit(X_train_features, Y_train)

prediction_on_test_data = gb_model.predict(X_test_features)

accuracy_on_test_data = accuracy_score(Y_test, prediction_on_test_data)

print(f"Test Accuracy: {accuracy_on_test_data:.3f}")
```

Code for Confusion matrix and error rates

```
"""Calculate and display confusion matrix, FNR, and FPR."""

tn, fp, fn, tp = confusion_matrix(Y_test,
prediction_on_test_data).ravel()

fnr = fn / (fn + tp)

fpr = fp / (fp + tn)

print(f"False Negative Rate: {fnr:.3f}")

print(f"False Positive Rate: {fpr:.3f}")

print(f"False Positives: {fp}")

print(f"False Negatives: {fn}")

print(f"True Positives: {tp}")
```

Code for plotting ROC curve

```
"""Plot the ROC curve for model predictions."""


fpr_values, tpr_values, thresholds = roc_curve(Y_test,
probabilities_on_test_data)

auc_score = roc_auc_score(Y_test, probabilities_on_test_data)


plt.plot(fpr_values, tpr_values, label=f"ROC curve (AUC =
{auc_score:.2f})")

plt.plot([0, 1], [0, 1], 'k--')

plt.xlabel("False Positive Rate")

plt.ylabel("True Positive Rate")

plt.title("ROC Curve")
plt.legend(loc="lower right")

plt.show()
```