Loading the data

```
In [ ]: import pandas as pd
        import seaborn as sns
        import numpy as np
        from sklearn import preprocessing
        import numpy as np
        from sklearn import *
        import matplotlib.pyplot as plt
```

```
In [ ]: from google.colab import drive
        drive.mount('/content/gdrive')
```

```
In [ ]: file_path = '/content/gdrive/MyDrive/Capstone/final_dataset.csv'
        df=pd.read_csv(file_path,)
        df
```

```
In [ ]: df2=df.drop(['index', 'Title',  'Artist','track_id'], axis=1)
        df2
        X = df2.drop(['Popular'], axis=1)
        y = df2['Popular']
```

Ensemble Methods

Balancing the data

We have a lot more unpopular data than popular. We want to balance that dataset before modeling. We are doing that for the following reasons:

- Mitigates bias: Imbalanced classes can bias the model towards the unpopular dataset. Balancing creates a more even distribution that allows the model to learn evenly from all classes.
- Makes training more stable: Imbalanced classes can lead to unstable training and difficulty converging. Balancing classes helps stabilize and optimize the training process especially for Logistic Regression which assumes an even class distribution, can be biased toward majority class otherwise, Decision trees also tend to overfit the majority class and ignore the minority class with skewed data. This applies for many more other models like Naive Bayes and SVMs.

We use the RandomOverSampler from Sklearn. These are the reasons we chose it:

- To balance an imbalanced dataset. RandomOverSampler will randomly replicate the popular class examples to increase their representation to be more balanced with the unpopular class. This can help improve model performance on the minority class.
- No loss of data. Undersampling will reduce the number of unpopular class examples, deleting potentially useful training data. Oversampling balances classes while retaining all data
- Maintain variability. Undersampling can lose rare but important samples from the unpopular class. Oversampling maintains the diversity of examples
- Require less data preprocessing. Oversampling is a relatively simple data augmentation technique. Undersampling needs more preprocessing to selectively reduce data

```
In [ ]: #Let us try with the standard scaler
        from imblearn.over_sampling import RandomOverSampler

        ros = RandomOverSampler(sampling_strategy="not majority")
        X_r, y_res = ros.fit_resample(X, y)
```

Scaling

The features are measured differently. For example danceability ranges between 0 and 1 while Loudness is between -60 and 0.Since our data has different ranges and are in different shapes, let us unify them by scaling

We are scaling for several the following reasons:

- Normalize feature values: Features like duration, loudness, and view count have very different scales. Scaling puts them on a common scale for fair comparison.
- Improve model performance: Scaling danceability, acousticness, tempo and other features helps optimization algorithms converge faster and more stably during model training.
- Reduce bias from outliers: Outliers in features like YouTube view count can skew results. Scaling minimizes their impact.

- Enable use of distance-based algorithms: Algorithms relying on distance measures require scaling of features like tempo and duration to properly measure distances.

We are using StandardScaler from Sklearn to scale the data. The reasons for this choice are:

- Centering and scaling: StandardScaler centers features like energy, loudness, and tempo to mean 0 and scales them to unit variance.
- Handles outliers well: Scaling to unit variance reduces the influence of outliers in YouTube view count compared to other scalers.
- Preserves shape: StandardScaler does not distort differences and correlations between features like acousticness and danceability.
- Efficient for large data: StandardScaler uses statistics computed on the full dataset making it efficient for scaling our large music data.

In [ ]:
```
stsc=preprocessing.StandardScaler()
X_res = stsc.fit_transform(X_r)
```

Modeling

1. Baseline Model: Logistic Regression

For the Logistic regression model that is our base model, the dataset is not too small, so a single 70-30 train-test split may be sufficient for baseline models

In [ ]:
```
#splitting data into train and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
     X_res, y_res, test_size=0.30, random_state=42)
```

For our classification problem, where many independent features predict one categorical binary output, we choose a logistic regression model as our baseline. The model is simple and stable, which makes it a good starting point for comparison with other models. Logistic regression model is fast in training and prediction, which will make it easier for us to rerun the model if we decide to create more itrations with new parameters or ensemble methods to it.

In [ ]:
```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
models=[]
lr = LogisticRegression()
lr.fit(X_train, y_train)
models.append(lr)
score = lr.score(X_test, y_test)
```

In [ ]:
```
# make predictions
predictions = lr.predict(X_test)

# get the performance metrics
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)

print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

Based on the performance results, we can see that logistic regression model have quite good results in prediction task. The model predicts 82,84% of the results correctly. Model is effective at avoiding false positives, because 82,97% of all positive are true positives. Model showed a relatively good preformance at avoding false negatives (80,04%). According to F1-score, our model have a high overall performance too(81,48%).

2. Random Forest Classifier

The second model that we used to consider for our classification problem is Random Forest. The model can be used for both regression and classification problems, but we can benefit from using it in our project due to several reasons. Compared to Decision Tree algorithm, Random Forest can give us more accurate results, since it uses a set of deicision trees to make a final classification prediction. Along with that, Random Forest model can prevent us from overfitting. Random Forest algorithm is well-fit for problems that include different types of features (in our case, we have audio features from Spotify, some of which categorical and some of which are continuous).

Let us explain a little bit the parameters chosen for the Random forest classifier:

1. *n_estimators:*
    - It determines the number of trees in the forest.
    - Choosing 500 trees can generally provide a more robust model that might improve prediction accuracy and generalization. looking at the size of our data, 500 seemmed to be a fair choice.

1. *max_depth:*
    - It limits the maximum depth of individual trees.
    - Setting `max_depth = 4` constrains the depth of trees to a relatively shallow level. This prevents the trees from becoming overly complex and reduces the risk of overfitting, especially for our dataset with a smaller number of features.

1. *max_features:*
    - It determines the maximum number of features considered for each split.
    - Setting `max_features = 3` limits the number of features considered for each split, which promotes diversity among the trees and reduces the risk of overfitting. This number is low to improve generalization.

1. *bootstrap:*
    - It decides whether to use bootstrap samples (random sampling with replacement) when building trees.
    - Setting `bootstrap = True` introduces randomness by using bootstrapping, contributing to the diversity of the trees and creating different subsets of data for each tree. This randomness improves the overall stability of the model.

1. *random_state:*
    - It controls the random number generation for reproducibility.
    - Setting `random_state = 18` ensures reproducibility, allowing the results to be consistent across different runs of the model. This parameter ensures that the random processes in the model remain the same, aiding in debugging and sharing the code.

In [ ]:
```
# import
from sklearn.ensemble import RandomForestClassifier

# start a model with 500 decision trees
rf = RandomForestClassifier(n_estimators = 500, max_depth = 4, max_features = 3, bootstrap = True, random_state

# train the model
rf.fit(X_train, y_train)
models.append(rf)
```

Let's evaluate the model:

In [ ]:
```
# make predictions
predictions = rf.predict(X_test)

# get the performance metrics
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)

print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

For the rest of our models, we will use cross-validation methos to enrich the data. This is because the dataset here is too small and we might risk overfitting without cross-validation.

Using 5 folds to get a balance between bias/variance. More folds like 10 can be used for larger data.

- Shuffling the data before creating folds ensures randomness.
- Setting a random state ensures reproducibility.
- Tracking accuracy per fold shows variability in performance.
- Random forest hyperparameters (like 100 trees, depth 5) can be tuned.
- Average final score gives overall performance estimate.

In [ ]:
```
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(rf, X_res, y_res, cv=5, scoring='accuracy')

for i, score in enumerate(cv_scores):
    print(f"Fold {i + 1}: {score}")
print(f"\nMean Accuracy: {cv_scores.mean()}")
```

According to the performance results, we can see that the random forest model shows good results in prediction task. The model predicts 84,02% of the results correctly. Model is effective at avoiding false positives, because 78,09% of all positive are true positives. Model showed a relatively good preformance at avoding false negatives (91,89%). According to F1-score, our model have a high overall performance too (84,43%). After cross-validation, the model got even more accurate and reached 85,46% accuracy in results.

3. XG Boost

The next model we decided to consider for our project is XG Boost. This algorithms is known for high speed and performance. XG Boost deals effectively with complex relationships between features, as well as non-linear relationships (in our project, the audio features and Youtube statistics do not follow a linear pattern with an outcome variable, and our independent features might have certain impact on one another making it more complex).

In [ ]:
```
#installing
!pip install xgboost
```

In [ ]:
```
#imports
from numpy import loadtxt
from xgboost import XGBClassifier
```

In [ ]:
```
#setting parameters for data splitting
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test_size=test_size, random_state=seed)
```

In [ ]:
```
# training model
xgb = XGBClassifier()
xgb.fit(X_train, y_train)
```

In [ ]:
```
#making predictions
y_pred = xgb.predict(X_test)
predictions = [round(value) for value in y_pred]
```

In [ ]:
```
# get the performance metrics
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1 = f1_score(y_test, predictions)

print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

We are using 10 folds here because: More data for training: With 10-folds, each fold uses ~90% of the data for training, while 5-folds uses ~80%. More training data helps complex models like ensembles generalize better

In [ ]:
```
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(xgb, X_res, y_res, cv=10, scoring='accuracy')

for i, score in enumerate(cv_scores):
    print(f"Fold {i + 1}: {score}")

print(f"\nMean Accuracy: {cv_scores.mean()}")
```

Based on the model performance, we can see that the XG Boost model performs well in our prediction task. The model predicts 87,70% of the results correctly. Model is effective at avoiding false positives, because 85,5% of all positive are true positives. Model showed a relatively good preformance at avoding false negatives (91,21%). According to F1-score, our model have a high overall performance too (88,27%). Using 10-cross-validation made our model more accurate, and as a result its accuracy reached 87,84%.

Other models explaination

SVC (Support Vector Classifier) was a potential machine learning model to try on this song popularity prediction task, but it may not be the best choice compared to some other options: Typically doesn't perform as well as tree or ensemble methods like random forests/XGBoost on tabular data Harder to tune kernel parameters compared to neural network hyperparameters Doesn't handle mixed numerical/categorical features as naturally as other models 1500 samples may be on the lower side for effectively training an SVC

KNN doesn't generalize well - it memorizes the training data. Performance is highly dependent on getting the value of K right. KNN is prone to overfitting, especially with numeric features like danceability, energy etc. Regularization is not easy. KNN doesn't handle irrelevant features well. It uses all features equally in distance calculation. KNN calculation cost grows with more training data. Prediction can get slow. Choosing distance metrics and weights on features is an added tuning step. Performance degrades quickly with high dimensional data as distances become similar. Categorical features require special handling compared to tree models.