Link to Jupyter Notebook for this write-up:
https://github.com/mbuck86/Capstone1/blob/master/In-Depth%20Analysis/In-Depth%20Analysis%20Notebook.ipynb

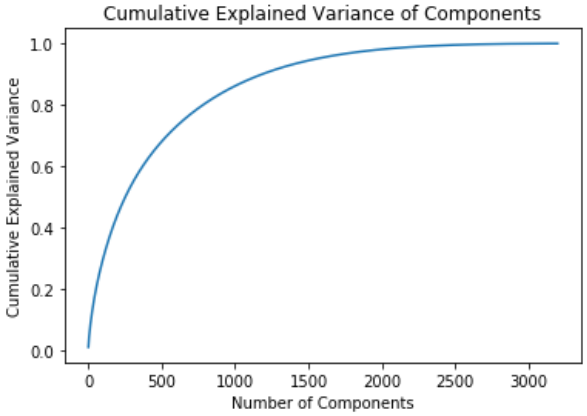
Capstone 1 In-Depth Analysis

The purpose of this document is to describe the steps taken to create the machine learning algorithm for my first Springboard capstone project. The objective of this algorithm will be to classify what decade a song is from based off of its lyrical content. The data that will be used for this algorithm was processed and cleaned earlier in the project. The columns of the data are words and the rows of the data are songs. The data are tf-idf values by song for each word that remained after data cleaning and filtering some nonsense words and stopwords from the data set. The songs in the dataset are the Billboard Top 100 songs for each year from 1965-2015, and the process of cleaning the data and exploratory analysis of the data was outlined earlier in the project.

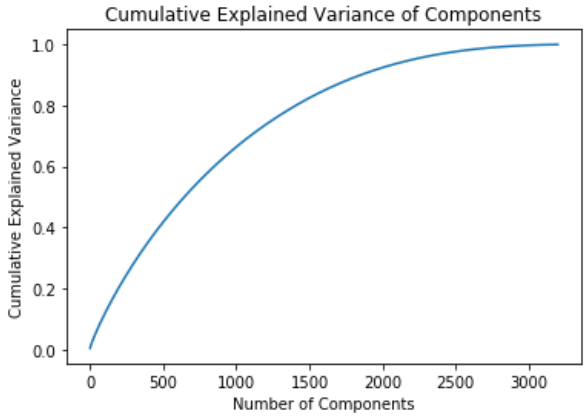
Since this problem is a classification problem, I started by choosing algorithms that are focused towards classification. The classifiers that I decided to try to use to address this problem were: A support vector machine classifier, a random forest classifier, and an XGBoost classifier. Support vector machines and random forests were mentioned as standard classifiers while working through the Springboard curriculum, and the XGBoost classifier was recommended by my mentor. My mentor mentioned that I should start by looking at as many models as possible to start with, which is why I chose to start with three as opposed to focusing on just one.

I first started by testing the out-of-the-box versions of each classifier on the tf-idf data. This was to get a general idea of how each classifier performed on the 'raw' tf-idf data in order to give myself a frame of reference as to what might be the best algorithm to use for the final product of the project. Each model was generated using their respective default parameters except for setting a random state for each model in order to ensure reproducibility. For these initial models data was split into a 80% training / 20% testing split, and initially nothing was addressed in the split for the unbalanced classes. The support vector machine classifier was the first model that I tested, which had a training accuracy of approximately 20% and a testing accuracy around 19%. In both cases these models are only slightly better than random guessing since there are six decades in the data set. The out-of-the-box random forest classifier performed significantly better with a training accuracy of 98.5% and a testing accuracy of 33%, which seems to demonstrate strong overfitting of the random forest classifier before parameter tuning. Finally the XGBoost classifier out of the box had a training accuracy of 69.7% and a testing accuracy of 37.5%. Out of our choices of model, it does seem like the XGBoost classifier will be the best classifier moving forward, however, I also checked the performance of each model after performing PCA on the features.

In an effort to improve model performance, I applied principal component analysis to the data in order to reduce the dimensionality of the features of the models. After performing PCA I selected a number of features from the primary components and ran each model again, selecting numbers of primary components based off of their explained variance. For this first PCA step, I chose the number of primary components to check of 1000, 1500, and 200, based off of the capture variance of those numbers of features, which can be observed in the following visualization. Those specific numbers seemed to be where the increase in captured explained variance started to decrease with the number of components.



The results of the models after this PCA were somewhat disappointing to say the least. No number of primary components improved the accuracy of the SVM model, and the testing accuracy of the random forest model decreased significantly to around 25% regardless of number of components. The XGBoost model had the most interesting results after PCA, as the training accuracy increased significantly to approximately 90%, but testing accuracy decreased to around 35%. After discussing these results with my mentor, I decided on two things. The first was that I was going to try using sklearn's StandardScaler in order to normalize my tf-idf values, and that I would focus on the XGBoost classifier moving forward, since it outperformed the other data in testing scores throughout the entirety of this analysis.



After standardizing the data, I used to out-of-the-box XGBoost classifier again in order to get an idea of its performance on the normalized data. The performance on the normalized data was identical to the performance on the non-normalized data, which was interesting to see. I then did PCA on the normalized data(captured cumulative variance of this new PCA to the right) and tested out a number of components to see what would give the best testing accuracy. With 1750 primary components, the model had a training accuracy of 87.5% and a testing accuracy of 37.5%. With this context, I was comfortable moving to parameter tuning for the model. The first parameter that I started tuning was the number of features. I used 5-fold cross validation to check average accuracy and standard deviation My mentor suggested that I use RandomizedSearchCV from the sklearn library, as it can be just as effective as GridSearchCV without checking every single aspect of the parameters every time, reducing calculation time. I first checked the range of

number of features which captured the best training accuracy in order to find the best number of features, I did this by checking the ranges between 1500 and 2000 features in steps of 25. The numbers of features which produced the best training accuracy were: 1575, 1600, 1700, and 1800. I then used cross validation to check the performance of each of those numbers of features with the XGBoost classifier and found that 1700 features produced the best accuracy through cross validation, so that is the number of features I decided to use.

I then moved forward with tuning the specific parameters of the model in order to try to improve the accuracy of the model. The features that I walked through tuning were: number of estimators, maximum depth, minimum child weight, gamma, colsample_bytree, and reg_alpha. I went through a process of checking each of the parameters individually with a grid search, and when I felt comfortable with finding the best parameters through a grid search cross validation, for each of the parameters, I put the reasonable ranges for each parameter into a full randomized search in order to find the best combinations of parameters by searching in what appeared to be the most optimum range for the model. Unfortunately, after the arduous work that was the parameter tuning process for this model, I could not find a set of parameters that could outperform the out of the box model by any significant margin. The final tuned parameters of the model were able to outperform the out of the box model on training accuracy, by taking the training accuracy up to 99.35% from 87.2% before parameter tuning, but those tuned parameters brought testing accuracy down to 35.6% from 38.2% from the out of the box model. After discussing this behavior with my mentor, he told me that this probably means that the model is pushed as far as it can go, and I am at the limits of what the model can do with these specific features. These results are somewhat disappointing to me but at this point in my education I have learned that most language is processed with neural networks which can consistently outperform the types of classifiers I have been working with. I will include the parameters for the models best features based on testing accuracy and log-loss, as the model optimized for log-loss ended up with at better training accuracy than the model optimized for simply for accuracy. The log-loss optimized model ended up with a testing accuracy of 37.38%, which is still not better than the out-of-the-box model, but is not that much worse.

In conclusion, it seems that this specific solution to this problem is definitely not the best. This solution is better than random guessing, as with six different classes random guessing would have an accuracy of around 16.67%, so the models all have accuracies two times better than random, but not anything that would necessarily be considered useful if we really cared about being accurate in the solving of this problem. The following images are the best parameters of the final log-loss optimized and accuracy optimized models, which were discussed above.

In [171]:
```
#log-loss optimized model
fit_f.best_params_
```

Out[171]:
```
{'subsample': 0.7,
 'reg_alpha': 0.01,
 'n_estimators': 150,
 'min_child_weight': 6,
 'max_depth': 3,
 'colsample_bytree': 0.7}
```

In [172]:
```
#accuracy optimized model
fit_facc.best_params_
```

Out[172]:
```
{'subsample': 0.85,
 'reg_alpha': 0.01,
 'n_estimators': 75,
 'min_child_weight': 4,
 'max_depth': 6,
 'colsample_bytree': 0.6}
```

:
```
#log-loss optimized training accuracy
log_tr = sum(trlog_pred==ytr)/len(ytr)
log_tr
```

: 0.9530927835051546

:
```
#log-loss optimized testing accuracy
log_te = sum(telog_pred==yte)/len(yte)
log_te
```

: 0.37384140061791965

:
```
# Accuracy optimized training and testing accuracy
tr_acc_pred = fit_facc.predict(Xtr)
```

:
```
te_acc_pred = fit_facc.predict(Xte)
```

:
```
acc_tr = sum(tr_acc_pred==ytr)/len(ytr)
acc_tr
```

: 0.9935567010309279

:
```
acc_te = sum(te_acc_pred==yte)/len(yte)
acc_te
```

: 0.35633367662203913