SVM and Logistic Regression Modeling

Created by Phillip Efthimion, Scott Payne, Gino Varghese and John Blevins

MSDS 7331 Data Mining - Section 403 - Mini Project

In this notebook a publicly available data set which is related to the popularities of online news sites will be analyzed. The online news popularity data set utilized in this analysis is publicly accessible from the UCI machine learning repository at https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity (https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity). The dataset can provide interesting insights into the popularity of online news articles based on article metadata such as word patterns, media content, message sentiment, day of publication and more. In this analysis, the ability to predict popularity is examined based on these attributes. As part of the analysis two different classification models will be considered: Logistic Regression and Support Vector Machines. Of particular interest is a comparison between these two classification techniques and interpretation of the results. The effectiveness of what makes either a good prediction algorithm will be based primarily on the performance accuracy as well as training time. In addition the input parameters for fitting each model will be varied to observe the affect on the output.

Python data tools will be used to tidy the data, fit the models and perform further exploratory analysis. The following Python modules are required:

- Pandas
- Numpy
- Matplotlib
- Seaborn
- Sklearn
- Plotly
- Statsmodels.api
- Warnings
- Datetime

The shares attribute in the dataset represents the popularity value to be estimated. This attribute is a continuous numeric type in the data set and must be recoded as a categorized response variable before performing the analysis. This is done by establishing threshold for "popular" versus "non-popular" and generating a new result variable where share values are grouped as such. The data will also be filtered based on articles having to do with technology. This is due to the size of the original dataset and the amount of processing required to fit models for the dataset in it's entirety. The population of interest is therefore limited to a these types of articles and any statistical results of the analysis limited as such. In addition, attributes which don't provide useful explanetory functions in the anaylsys will be removed from the dataset. An 80/20 training/testing split will be utilized for the model fitting and each model will be assessed for accuracy and performance. Given the relatively small number of features in the data set a linear kernel specification will be used to fit each model.

In [173]:
```
# Import and Configure Required Modules
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns
import warnings
import datetime
warnings.simplefilter('ignore', DeprecationWarning)
plt.rcParams['figure.figsize']=(15,10)

# Read Online News Data
df = pd.read_csv('data/OnlineNewsPopularity.csv')

# Correct Column Names by Removing Leading Space
df.columns = df.columns.str.replace(' ', '')

# Rename Columns for Ease of Display
df = df.rename(columns={'weekday_is_monday': 'monday', 'weekday_is_tuesday': 'tuesday', 'weekday_
is_wednesday': 'wednesday', 'weekday_is_thursday': 'thursday', 'weekday_is_friday': 'friday', 'we
ekday_is_saturday': 'saturday', 'weekday_is_sunday': 'sunday', 'is_weekend': 'weekend'})
df = df.rename(columns={'data_channel_is_lifestyle':'lifestyle',
'data_channel_is_entertainment':'entertainment', 'data_channel_is_bus':'business', 'data_channel_
is_socmed':'social_media', 'data_channel_is_tech':'technology', 'data_channel_is_world':'world'})

# Encode a new "popular" column based on the # of shares
# "popular" = 1 and "not popular" to 0.
df['popularity'] = pd.qcut(df['shares'].values, 2, labels=[0,1])
df.popularity = df.popularity.astype(np.int)

# Take a subset of the data related to Technology News Articles
dfsubset = df.loc[df['technology'] == 1]

# Reassign to New Variable and remove Columns which aren't needed
```

```
df_imputed = dfsubset
del df_imputed['url']
del df_imputed['shares']
del df_imputed['timedelta']
del df_imputed['lifestyle']
del df_imputed['entertainment']
del df_imputed['business']
del df_imputed['social_media']
del df_imputed['technology']
del df_imputed['world']

# Display Dataframe Structure
df_imputed.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7346 entries, 4 to 39639
Data columns (total 53 columns):
n_tokens_title                 7346 non-null float64
n_tokens_content               7346 non-null float64
n_unique_tokens                7346 non-null float64
n_non_stop_words               7346 non-null float64
n_non_stop_unique_tokens       7346 non-null float64
num_hrefs                      7346 non-null float64
num_self_hrefs                 7346 non-null float64
num_imgs                       7346 non-null float64
num_videos                     7346 non-null float64
average_token_length           7346 non-null float64
num_keywords                   7346 non-null float64
kw_min_min                     7346 non-null float64
kw_max_min                     7346 non-null float64
kw_avg_min                     7346 non-null float64
kw_min_max                     7346 non-null float64
kw_max_max                     7346 non-null float64
kw_avg_max                     7346 non-null float64
kw_min_avg                     7346 non-null float64
kw_max_avg                     7346 non-null float64
kw_avg_avg                     7346 non-null float64
self_reference_min_shares      7346 non-null float64
self_reference_max_shares      7346 non-null float64
self_reference_avg_sharess     7346 non-null float64
monday                         7346 non-null float64
tuesday                        7346 non-null float64
wednesday                      7346 non-null float64
thursday                       7346 non-null float64
friday                         7346 non-null float64
saturday                       7346 non-null float64
sunday                         7346 non-null float64
weekend                        7346 non-null float64
LDA_00                         7346 non-null float64
LDA_01                         7346 non-null float64
LDA_02                         7346 non-null float64
LDA_03                         7346 non-null float64
LDA_04                         7346 non-null float64
global_subjectivity            7346 non-null float64
global_sentiment_polarity      7346 non-null float64
global_rate_positive_words     7346 non-null float64
global_rate_negative_words     7346 non-null float64
rate_positive_words            7346 non-null float64
rate_negative_words            7346 non-null float64
avg_positive_polarity          7346 non-null float64
min_positive_polarity          7346 non-null float64
max_positive_polarity          7346 non-null float64
avg_negative_polarity          7346 non-null float64
min_negative_polarity          7346 non-null float64
max_negative_polarity          7346 non-null float64
title_subjectivity             7346 non-null float64
title_sentiment_polarity       7346 non-null float64
abs_title_subjectivity         7346 non-null float64
abs_title_sentiment_polarity   7346 non-null float64
popularity                     7346 non-null int32
dtypes: float64(52), int32(1)
memory usage: 3.0 MB
```

Training and Testing Split

The binary popularity value will be predicted, where 0 represents non-popular articles while 1 represents popular articles. The remaining attributes

The binary popularity value will be predicted, where 0 represents non popular articles while 1 represents popular articles. The remaining attributes will be included in the development of the prediction models. An instance of the ShuffleSplit object is created from sklearn to setup the cross validation parameters. The data will be split such that 80% of the data is put into a training dataset and the remaining 20% in a test dataset.

In [174]:
```
from sklearn.model_selection import ShuffleSplit

# we want to predict the X and y data as follows:
if 'popularity' in df_imputed:
    y = df_imputed['popularity'].values # get the labels we want
    del df_imputed['popularity'] # get rid of the class label
    X = df_imputed.values # use everything else to predict!

    ## X and y are now numpy matrices, by calling 'values' on the pandas data frames we
    #    have converted them into simple matrices to use with scikit learn


# to use the cross validation object in scikit learn, we need to grab an instance
#    of the object and set it up. This object will be able to split our data into
#    training and testing splits, 3 iterations
#    80% in training dataset and 20% for test
num_cv_iterations = 3
num_instances = len(y)
cv_object = ShuffleSplit(n_splits=num_cv_iterations,
                         test_size  = 0.2)

print(cv_object)
```

```
ShuffleSplit(n_splits=3, random_state=None, test_size=0.2, train_size=None)
```

Logistic Regression

A logistic regression model will be fit on the three iterations of the cross validation object (cv_object). During each iteration the resultant accuracy, confusion matrix and training time will be calculated.

In [175]:
```
# run logistic regression and vary some parameters
from sklearn.linear_model import LogisticRegression
from sklearn import metrics as mt

# first we create a reusable logisitic regression object
#   here we can setup the object with different learning parameters and constants
lr_clf = LogisticRegression(penalty='l2', C=1.0, class_weight=None) # get object

# now we can use the cv_object that we setup before to iterate through the
#    different training and testing sets. Each time we will reuse the logisitic regression
#    object, but it gets trained on different data each time we use it.

iter_num=0
# the indices are the rows used for training and testing in each iteration
for train_indices, test_indices in cv_object.split(X,y):
    # I will create new variables here so that it is more obvious what
    # the code is doing (you can compact this syntax and avoid duplicating memory,
    # but it makes this code less readable)
    X_train = X[train_indices]
    y_train = y[train_indices]

    X_test = X[test_indices]
    y_test = y[test_indices]

    # train the reusable logisitc regression model on the training data
    train_start_time = datetime.datetime.now()
    lr_clf.fit(X_train,y_train)  # train object
    train_end_time = datetime.datetime.now()
    train_time = train_end_time - train_start_time

    y_hat = lr_clf.predict(X_test) # get test set precitions

    # now let's get the accuracy and confusion matrix for this iterations of training/testing
    acc = mt.accuracy_score(y_test,y_hat)
    conf = mt.confusion_matrix(y_test,y_hat)
    print("====Iteration",iter_num," ====")
    print("accuracy", acc )
    print("Confusion matrix\n",conf)
    print("Training time", train_time)
    iter_num+=1


# Also note that every time you run the above code
#   it randomly creates a new training and testing set,
```

```
#   so accuracy will be different each time
```

```
====Iteration 0  ====
accuracy 0.613605442177
Confusion matrix
 [[105 476]
 [ 92 797]]
Training time 0:00:00.353237
====Iteration 1  ====
accuracy 0.612925170068
Confusion matrix
 [[122 500]
 [ 69 779]]
Training time 0:00:00.181108
====Iteration 2  ====
accuracy 0.61156462585
Confusion matrix
 [[130 475]
 [ 96 769]]
Training time 0:00:00.223217
```

Parameter Adjustment for Improving Accuracy

Attributes that don't provide any explanatory value have already been removed that would have affected the fitted model. In addition the cost can be adjusted to improve accuracy. The cost parameter tells the model optimization how much training data you want to avoid being misclassified. The large the cost, the optimization will choose smaller-margin hyperplane, the hyperplane does a good job in getting all the points in the training data classified. Similarly a small cost value will let the optimization to look for larger-margin separating hyperplane,in this approach the hyperplane ignore some points the training data will still be linearly separable.

In [176]:
```
# here we can change some of the parameters interactively
from ipywidgets import widgets as wd

def lr_explor(cost):
    lr_clf = LogisticRegression(penalty='l2', C=cost, class_weight=None) # get object
    accuracies = cross_val_score(lr_clf,X,y=y,cv=cv_object) # this also can help with parallelism
    print(accuracies)

wd.interact(lr_explor,cost=(0.001,5.0,0.05),__manual=True)
```

Out[176]:
```
<function __main__.lr_explor>
```

Interpretting weights for Logistic Regression

The weights of the coefficients for logistic regressions are important for determining what attributes to include in the model. The logistic regression coefficients are used to predict the probability of an outcome, in this case, the popularity of the news article being "popular" or "not popular". A positive weight indicates that an increase in the attribute will increase the odds of the outcome being “popular”, while a negative weight indicates that an increase in the attribute will decrease the likelihood of the outcome being “popular”.

In [177]:
```
# interpret the weights

# iterate over the coefficients
weights = lr_clf.coef_.T # take transpose to make a column vector
variable_names = df_imputed.columns
for coef, name in zip(weights,variable_names):
    print(name, 'has weight of', coef[0])

# does this look correct?
```

```
n_tokens_title has weight of -7.10792904898e-05
n_tokens_content has weight of 0.00045292972299
n_unique_tokens has weight of -4.09849637017e-06
n_non_stop_words has weight of -5.60127922705e-06
n_non_stop_unique_tokens has weight of -4.39651089655e-06
num_hrefs has weight of 7.707342163e-05
num_self_hrefs has weight of -5.46430176411e-05
num_imgs has weight of -5.14384853249e-05
num_videos has weight of 3.23752597485e-06
average_token_length has weight of -2.54934573905e-05
num_keywords has weight of -3.49800869181e-05
kw_min_min has weight of -0.000306169585352
kw_max_min has weight of 7.01118936766e-05
kw_avg_min has weight of -0.000614081710732
kw_min_max has weight of -1.55765394436e-06
kw_max_max has weight of -1.46941861635e-06
kw_avg_max has weight of 2.25365206934e-07
```

```
kw_min_avg has weight of 5.57342602933e-05
kw_max_avg has weight of -4.4786246584e-05
kw_avg_avg has weight of 0.000537724917605
self_reference_min_shares has weight of 3.32789938055e-06
self_reference_max_shares has weight of 9.72691400289e-07
self_reference_avg_sharess has weight of -1.50963495668e-06
monday has weight of -5.08311199003e-06
tuesday has weight of -5.29205267136e-06
wednesday has weight of -7.17169975843e-06
thursday has weight of -5.26548160138e-06
friday has weight of 2.81724645072e-06
saturday has weight of 8.59357074472e-06
sunday has weight of 6.14746762413e-06
weekend has weight of 1.47410383689e-05
LDA_00 has weight of -1.26905194196e-08
LDA_01 has weight of -1.71731356553e-06
LDA_02 has weight of -1.21756372776e-06
LDA_03 has weight of 1.36028341004e-07
LDA_04 has weight of -2.44252172993e-06
global_subjectivity has weight of -1.80399862354e-06
global_sentiment_polarity has weight of -1.58721589632e-06
global_rate_positive_words has weight of -3.62964734743e-07
global_rate_negative_words has weight of 2.15821298205e-08
rate_positive_words has weight of -5.64789701934e-06
rate_negative_words has weight of 4.66177101334e-08
avg_positive_polarity has weight of -1.95737212952e-06
min_positive_polarity has weight of -1.06274688833e-06
max_positive_polarity has weight of -2.221795255e-06
avg_negative_polarity has weight of 3.78596440033e-07
min_negative_polarity has weight of -7.47071721552e-07
max_negative_polarity has weight of 1.3960281968e-06
title_subjectivity has weight of 4.12081719739e-07
title_sentiment_polarity has weight of 2.19420080019e-06
abs_title_subjectivity has weight of -1.11666551817e-06
abs_title_sentiment_polarity has weight of 2.17790026597e-06
```

Normalizing Features

Because the attributes do not all use the same measurement scale, the magnitude of the weights does not give meaningful insight into which attributes are most important for the model. The weights need to be scaled in the same way so that the magnitudes can be compared across all the features. The standard scaler will adjust the values of each attribute to be scaled by the average and standard deviation of each feature. The linear regression model will then be fit to the scaled values and the coefficient weights will be calculated based on that model.

In [178]:
```
from sklearn.preprocessing import StandardScaler

# we want to normalize the features based upon the mean and standard deviation of each column.
# However, we do not want to accidentally use the testing data to find out the mean and std (this
 would be snooping)
# to Make things easier, let's start by just using whatever was last stored in the variables:
##    X_train , y_train , X_test, y_test (they were set in a for loop above)

# scale attributes by the training set
scl_obj = StandardScaler()
scl_obj.fit(X_train) # find scalings for each column that make this zero mean and unit std
# the line of code above only looks at training data to get mean and std and we can use it
# to transform new feature data

X_train_scaled = scl_obj.transform(X_train) # apply to training
X_test_scaled = scl_obj.transform(X_test) # apply those means and std to the test set (without sn
ooping at the test set values)

# train the model just as before
lr_clf = LogisticRegression(penalty='l2', C=0.05) # get object, the 'C' value is less (can you gu
ess why??)
lr_clf.fit(X_train_scaled,y_train)  # train object

y_hat = lr_clf.predict(X_test_scaled) # get test set precitions

acc = mt.accuracy_score(y_test,y_hat)
conf = mt.confusion_matrix(y_test,y_hat)
print('accuracy:', acc )
print(conf )

# sort these attributes and spit them out
zip_vars = zip(lr_clf.coef_.T,df_imputed.columns) # combine attributes
zip_vars = sorted(zip_vars)
for coef, name in zip_vars:
    print(name, 'has weight of', coef[0]) # now print them out
```

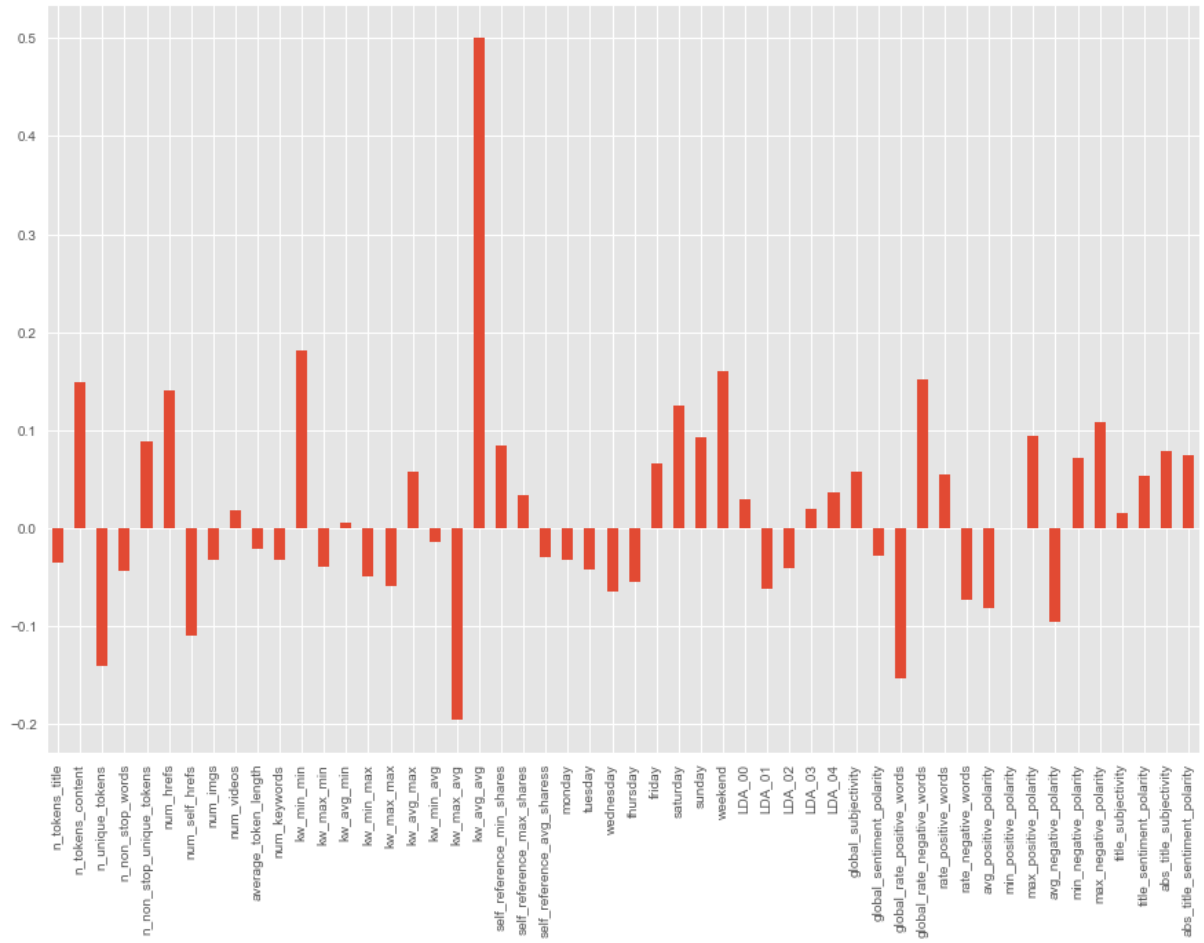
```
    print(name, 'has weight of', coef[0]) # now print them out
```

```
accuracy: 0.638775510204
[[251 354]
 [177 688]]
kw_max_avg has weight of -0.19511745499
global_rate_positive_words has weight of -0.152962693348
n_unique_tokens has weight of -0.140725072154
num_self_hrefs has weight of -0.109302767727
avg_negative_polarity has weight of -0.0963532717117
avg_positive_polarity has weight of -0.0821706245506
rate_negative_words has weight of -0.0730766216593
wednesday has weight of -0.0653846114312
LDA_01 has weight of -0.0624433372139
kw_max_max has weight of -0.0594930084952
thursday has weight of -0.0556334277959
kw_min_max has weight of -0.0491600585742
n_non_stop_words has weight of -0.0437150166118
tuesday has weight of -0.0421396679766
LDA_02 has weight of -0.0416606797504
kw_max_min has weight of -0.0402744070881
n_tokens_title has weight of -0.0354816838379
num_imgs has weight of -0.0330472884563
monday has weight of -0.0323374784223
num_keywords has weight of -0.0320350917896
self_reference_avg_sharess has weight of -0.0301987963171
global_sentiment_polarity has weight of -0.0289380941517
average_token_length has weight of -0.02060623972
kw_min_avg has weight of -0.014417762091
min_positive_polarity has weight of -0.000899705018967
kw_avg_min has weight of 0.00601961632871
title_subjectivity has weight of 0.0146850220318
num_videos has weight of 0.0180370921204
LDA_03 has weight of 0.0191275281479
LDA_00 has weight of 0.0287699434121
self_reference_max_shares has weight of 0.0338601200533
LDA_04 has weight of 0.0356840243098
title_sentiment_polarity has weight of 0.0536507616568
rate_positive_words has weight of 0.0545100979125
kw_avg_max has weight of 0.0568139549061
global_subjectivity has weight of 0.057576020448
friday has weight of 0.0661851944187
min_negative_polarity has weight of 0.0720667219016
abs_title_sentiment_polarity has weight of 0.0748442640363
abs_title_subjectivity has weight of 0.0791506833418
self_reference_min_shares has weight of 0.0840868577435
n_non_stop_unique_tokens has weight of 0.0885403332181
sunday has weight of 0.0924625521827
max_positive_polarity has weight of 0.0942286523379
max_negative_polarity has weight of 0.107880233359
saturday has weight of 0.12498011335
num_hrefs has weight of 0.140127506988
n_tokens_content has weight of 0.148998582074
global_rate_negative_words has weight of 0.152124356666
weekend has weight of 0.160243487053
kw_min_min has weight of 0.180532622697
kw_avg_avg has weight of 0.499865718975
```

The weights of the regression coefficients are now scaled and the magnitudes can be compared against each other. The greatest magnitude coefficients are features that contain information about the maximum, average, and minimum number of shares for the post's keyword (kw_max_avg, kw_min_min, kw_avg_avg). It makes sense that keywords with a higher minimum and average popularity would increase the probability of the outcome being popular. The number of links (num_hrefs) found within the article has a large positive magnitude, while the number of links to other articles on the same website(num_self_hrefs) has a large negative magnitude. It is possible that there is a collinearity problem with these features that should be explored further. The rate of positive words and rate of negative words have nearly identical magnitudes but in opposite directions, rate of positive words decreases the probability of popularity and rate of negative words increases it. The features that determine if the day is Saturday or the day is a weekend both have large positive magnitudes. Since these features are related, only including the feature indicating if the day is a weekend should not hurt the model. The last three impactful features are concerning the number of tokens, unique tokens, and nonstop unique tokens in the article. Greater number of tokens, or longer articles and nonstop unique tokens seem to increase the odds of popularity while total number of unique tokens is negative. This would indicate that longer articles with flourishes of colorful language can increase popularity while a large vocabulary in general can decrease popularity.

In [179]:
```
# now let's make a pandas Series with the names and values, and plot them
from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
plt.rcParams['figure.figsize']=(15,10)
```

```
weights = pd.Series(lr_clf.coef_[0],index=df_imputed.columns)
weights.plot(kind='bar')
plt.show()
```



Based on the graph of the coefficient weights, a cut off of > 0.1 and < - 0.1 would include the highest magnitude features while removing some of the less impactful attributes. Using this threshold, the following features would be included in the model: n_tokens_content, n_unique_tokens, n_non_stop_unique_tokens, num_hrefs, num_self_hrefs, kw_min_min, kw_max_max, kw_max_avg, kw_avg_avg, weekend, global_rate_positive_words, global_rate_negative_words.

In [180]:
```
from sklearn.preprocessing import StandardScaler
# we want to normalize the features based upon the mean and standard deviation of each column.
# However, we do not want to accidentally use the testing data to find out the mean and std (this
 would be snooping)

from sklearn.pipeline import Pipeline
# you can apply the StandardScaler function inside of the cross-validation loop
#  but this requires the use of PipeLines in scikit.
#  A pipeline can apply feature pre-processing and data fitting in one compact notation
#  Here is an example!

std_scl = StandardScaler()
lr_clf = LogisticRegression(penalty='l2', C=0.05)

# create the pipline
piped_object = Pipeline([('scale', std_scl),  # do this
                         ('logit_model', lr_clf)]) # and then do this

weights = []
# run the pipline cross validated
for iter_num, (train_indices, test_indices) in enumerate(cv_object.split(X,y)):
    piped_object.fit(X[train_indices],y[train_indices])  # train object
    # it is a little odd getting trained objects from a  pipeline:
    weights.append(piped_object.named_steps['logit_model'].coef_[0])


weights = np.array(weights)
```

In [181]:
```
import plotly
```

```
plotly.offline.init_notebook_mode() # run at the start of every notebook

error_y=dict(
            type='data',
            array=np.std(weights,axis=0),
            visible=True
        )

graph1 = {'x': df_imputed.columns,
          'y': np.mean(weights,axis=0),
    'error_y':error_y,
       'type': 'bar'}

fig = dict()
fig['data'] = [graph1]
fig['layout'] = {'title': 'Logistic Regression Weights, with error bars'}

plotly.offline.iplot(fig)
```

If we use a 0.1 threshold from the weight plot, we see that n_tokens_content, n_unique_tokens, n_non_stop_unique_tokens, num_hrefs, num_self_hrefs, kw_min_min, kw_max_max, kw_max_avg, kw_avg_avg, self_reference_avg_sharess, saturday, weekend, global_rate_positive_words, global_rate_negative_words and rate_negative_words all have the most weight in predicting popularity.

In [182]:
```
Xnew = df_imputed[['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens',
'num_hrefs', 'num_self_hrefs', 'kw_min_min', 'kw_max_max', 'kw_max_avg', 'kw_avg_avg', 'self_refe
rence_avg_sharess', 'saturday', 'weekend', 'global_rate_positive_words', 'global_rate_negative_wo
rds', 'rate_negative_words']].values

weights = []
# run the pipline corssvalidated
for iter_num, (train_indices, test_indices) in enumerate(cv_object.split(Xnew,y)):
    piped_object.fit(Xnew[train_indices],y[train_indices])  # train object
    weights.append(piped_object.named_steps['logit_model'].coef_[0])

weights = np.array(weights)

error_y=dict(
            type='data',
            array=np.std(weights,axis=0),
            visible=True
        )

graph1 = {'x': ['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens', 'num_hrefs',
'num_self_hrefs', 'kw_min_min', 'kw_max_max', 'kw_max_avg', 'kw_avg_avg', 'self_reference_avg_sha
ress', 'saturday', 'weekend', 'global_rate_positive_words', 'global_rate_negative_words', 'rate_n
egative_words'],
          'y': np.mean(weights,axis=0)
```

```
            'y': np.mean(weights,axis=0),
    'error_y':error_y,
       'type': 'bar'}

fig = dict()
fig['data'] = [graph1]
fig['layout'] = {'title': 'Logistic Regression Weights, with error bars'}

plotly.offline.iplot(fig)
```

If we use a 0.1 threshold from the weight plot, we see that n_tokens_content, n_unique_tokens, n_non_stop_unique_tokens, num_hrefs, num_self_hrefs, kw_min_min, kw_max_max, kw_max_avg, kw_avg_avg, weekend, global_rate_positive_words, global_rate_negative_words and rate_negative_words all have the most weight in predicting popularity.

In [183]:
```
Xnew = df_imputed[['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens',
'num_hrefs', 'num_self_hrefs', 'kw_min_min', 'kw_max_max', 'kw_max_avg', 'kw_avg_avg', 'weekend',
 'global_rate_positive_words', 'global_rate_negative_words', 'rate_negative_words']].values

weights = []
# run the pipline corssvalidated
for iter_num, (train_indices, test_indices) in enumerate(cv_object.split(Xnew,y)):
    piped_object.fit(Xnew[train_indices],y[train_indices])  # train object
    weights.append(piped_object.named_steps['logit_model'].coef_[0])

weights = np.array(weights)

error_y=dict(
            type='data',
            array=np.std(weights,axis=0),
            visible=True
        )

graph1 = {'x': ['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens', 'num_hrefs',
'num_self_hrefs', 'kw_min_min', 'kw_max_max', 'kw_max_avg', 'kw_avg_avg', 'weekend', 'global_rate
_positive_words', 'global_rate_negative_words', 'rate_negative_words'],
          'y': np.mean(weights,axis=0),
    'error_y':error_y,
       'type': 'bar'}

fig = dict()
fig['data'] = [graph1]
fig['layout'] = {'title': 'Logistic Regression Weights, with error bars'}

plotly.offline.iplot(fig)
```

If we use a 0.1 threshold from the weight plot, we see that n_tokens_content, n_unique_tokens, n_non_stop_unique_tokens, num_hrefs, num_self_hrefs, kw_min_min, kw_max_avg, kw_avg_avg, weekend, global_rate_positive_words, global_rate_negative_words and rate_negative_words all have the most weight in predicting popularity.

In [184]:
```
Xnew = df_imputed[['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens',
'num_hrefs', 'num_self_hrefs', 'kw_min_min', 'kw_max_avg', 'kw_avg_avg', 'weekend', 'global_rate_
positive_words', 'global_rate_negative_words', 'rate_negative_words']].values

weights = []
# run the pipline corssvalidated
for iter_num, (train_indices, test_indices) in enumerate(cv_object.split(Xnew,y)):
    piped_object.fit(Xnew[train_indices],y[train_indices])  # train object
    weights.append(piped_object.named_steps['logit_model'].coef_[0])

weights = np.array(weights)

error_y=dict(
            type='data',
            array=np.std(weights,axis=0),
            visible=True
        )

graph1 = {'x': ['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens', 'num_hrefs',
'num_self_hrefs', 'kw_min_min', 'kw_max_avg', 'kw_avg_avg', 'weekend', 'global_rate_positive_word
s', 'global_rate_negative_words', 'rate_negative_words'],
          'y': np.mean(weights,axis=0),
    'error_y':error_y,
       'type': 'bar'}

fig = dict()
fig['data'] = [graph1]
fig['layout'] = {'title': 'Logistic Regression Weights, with error bars'}

plotly.offline.iplot(fig)
```

Support Vector Machines

A support vector machine model will be fit on the three iterations of the cross validation object (cv_object). During each iteration the resultant accuracy, confusion matrix and training time will be calculated.

Parameter Adjustment for Improved Accuracy

Either Linear or RBF kernel could have been used for this analysis however the linear kernel tends to perform better with a larger number of features. RBF would be more appropriate for other applications like image classification.

In [185]:
```
from sklearn.svm import SVC

iter_num=0
# okay, so run through the cross validation loop and set the training and testing variable for on
e single iteration
for train_indices, test_indices in cv_object.split(X,y):
    # I will create new variables here so that it is more obvious what
    # the code is doing (you can compact this syntax and avoid duplicating memory,
    # but it makes this code less readable)
    X_train = X[train_indices]
    y_train = y[train_indices]

    X_test = X[test_indices]
    y_test = y[test_indices]

    X_train_scaled = scl_obj.transform(X_train) # apply to training
    X_test_scaled = scl_obj.transform(X_test)

    # train the model just as before
    svm_clf = SVC(C=1, kernel='linear', degree=3, gamma='auto') # get object
    #svm_clf = SVC(C=0.5, kernel='rbf', degree=3, gamma='auto') # get object
    train_start_time = datetime.datetime.now()
    svm_clf.fit(X_train_scaled, y_train)  # train object
    train_end_time = datetime.datetime.now()

    train_time = train_end_time - train_start_time

    y_hat = svm_clf.predict(X_test_scaled) # get test set precitions

    acc = mt.accuracy_score(y_test,y_hat)
    conf = mt.confusion_matrix(y_test,y_hat)

    print("====Iteration",iter_num," ====")
    print('Accuracy:', acc )
    print('Training time:', train_time)
    print('Confusion Matrix:', conf)

    iter_num+=1
```

```
====Iteration 0  ====
Accuracy: 0.637414965986
Training time: 0:00:06.555547
Confusion Matrix: [[169 400]
 [133 768]]
====Iteration 1  ====
Accuracy: 0.644897959184
Training time: 0:00:06.266684
Confusion Matrix: [[190 406]
 [116 758]]
```

```
 [110 758]]
====Iteration 2  ====
Accuracy: 0.619727891156
Training time: 0:00:06.181424
Confusion Matrix: [[182 411]
 [148 729]]
```

The accuracy of the support vectors for iteration zero is .6394. This means that 64% of the vectors are correctly classified out of a total of 1470. According to the confusion matrix, there were 162 predicted no that were actually no. There were 420 that were predicted yes, but were actually no. There were 110 that were predicted no, but were actually yes, and there were 778 that were predicted yes and were actually yes. Therefore, there were 940 predicted correctly and 530 predicted incorrectly.

The accuracy of the support vectors for iteration one is .6326. This means that 63% of the vectors are correctly classified out of a total of 1470. According to the confusion matrix, there were 174 predicted no that were actually no. There were 412 that were predicted yes, but were actually no. There were 128 that were predicted no, but were actually yes, and there were 756 that were predicted yes and were actually yes. Therefore, there were 930 predicted correctly and 540 predicted incorrectly.

The accuracy of the support vectors for iteration 2 is .6149. This means that 61% of the vectors are correctly classified out of a total of 1470. According to the confusion matrix, there were 163 predicted no that were actually no. There were 436 that were predicted yes, but were actually no. There were 130 that were predicted no, but were actually yes, and there were 741 that were predicted yes and were actually yes. Therefore, there were 904 predicted correctly and 566 predicted incorrectly.

The dimensions of the array of support vectors, the dimensions of the indices and the number of support vectors for each class can be displayed.

In [186]:
```
# look at the support vectors
print(svm_clf.support_vectors_.shape)
print(svm_clf.support_.shape)
print(svm_clf.n_support_ )
```

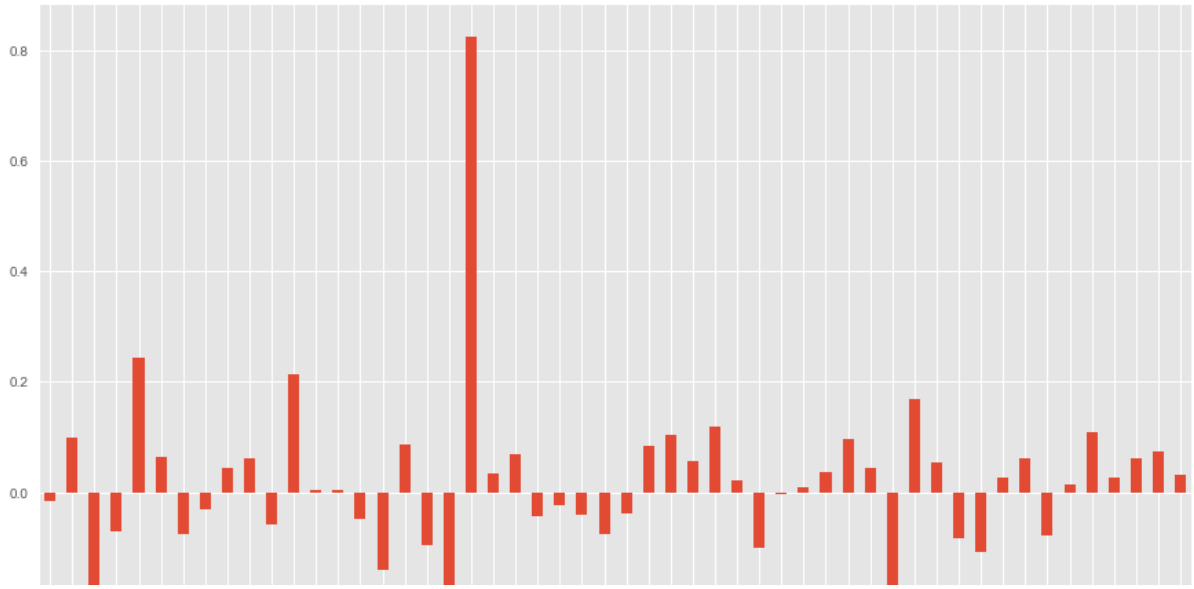
```
(4641, 52)
(4641,)
[2311 2330]
```

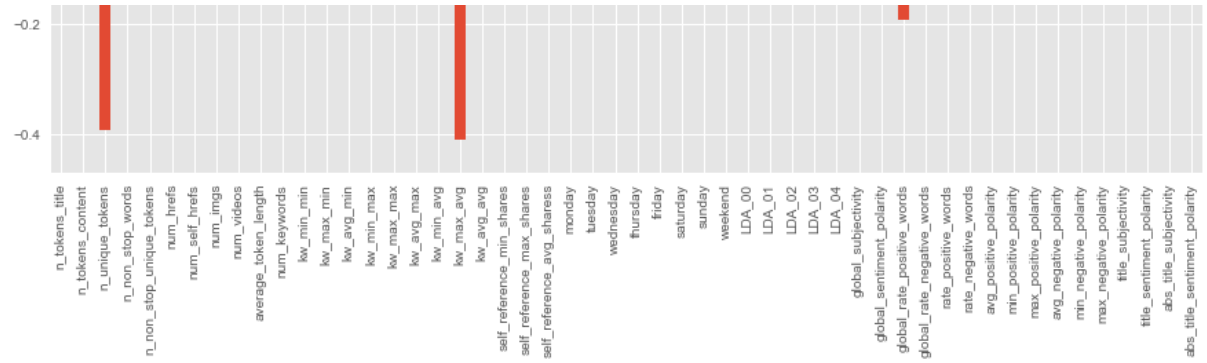
The dimensions of the arrays of the support vectors are (4637, 52). The dimensions of the indices of the support vectors are around (4637,) The number of support vectors for each class is [2309, 2328]. That is since 0 was classified as not popular and 1 was classified as popular, there are 2309 instances classified as not popular and 2328 instances classified as popular.

In [187]:
```
# if using linear kernel, these make sense to look at (not otherwise, why?)
print(svm_clf.coef_)
weights = pd.Series(svm_clf.coef_[0],index=df_imputed.columns)
weights.plot(kind='bar')
```

```
[[-0.01683534  0.09871504 -0.39132424 -0.07164146  0.24201198  0.06297425
  -0.07699087 -0.03003036  0.04273069  0.06129727 -0.0577065   0.21352547
   0.00270117  0.00301477 -0.04787386 -0.14025256  0.08616062 -0.0947733
  -0.40885313  0.82287889  0.0346819   0.06867362 -0.04471009 -0.02284498
  -0.0402815  -0.07526528 -0.03757991  0.08454724  0.10406922  0.05662065
   0.11943787  0.02132233 -0.10196862 -0.003588    0.00863082  0.03614872
   0.09721488  0.04478165 -0.193297    0.16777682  0.05329081 -0.08244257
  -0.10930966  0.02569783  0.06003511 -0.07879044  0.01323309  0.10826996
   0.02575798  0.06091245  0.0746941   0.03189972]]
```

Out[187]: <matplotlib.axes._subplots.AxesSubplot at 0x222886d2240>



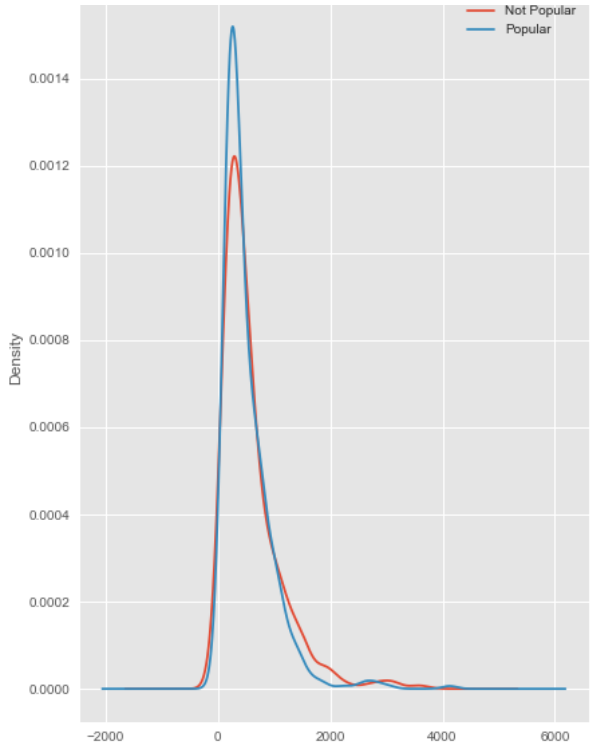


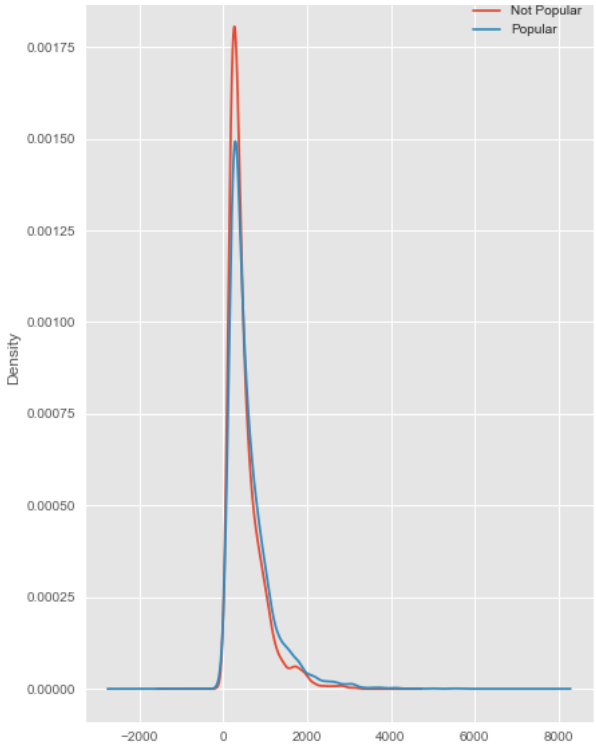
Next we have the weights of each support vector. Plotting them, we can see which weights are the strongest and the weakest. Classifying coefficients. Kw_avg_avg had the highest coefficient, which means that it has the strongest weight. Other significant coefficients are kw_max_avg, n_unique_tokens, n_non_stop_unique*tokens*, global_rate_positive_words, and global_rate_negative_words. Some of the support vectors that have the smallest coefficients are kw_avg_min, self_reference_max_shares, LDA_02, and title_subjectivity.

In [188]:
```
# Now let's do some different analysis with the SVM and look at the instances that were chosen as
 support vectors

# now lets look at the support for the vectors and see if we they are indicative of anything
# grabe the rows that were selected as support vectors (these are usually instances that are hard
 to classify)

# make a dataframe of the training data
df_tested_on = df_imputed.loc[train_indices] # saved from above, the indices chosen for training
# now get the support vectors from the trained model
df_support = df_tested_on.loc[svm_clf.support_,:]
df_support['popularity'] = y[svm_clf.support_] # add back in the 'Survived' Column to the pandas
 dataframe
#print(df_imputed['popularity'])

#df_imputed.info()
df_imputed['popularity'] = y # also add it back in for the original data
#df_support.info()
```

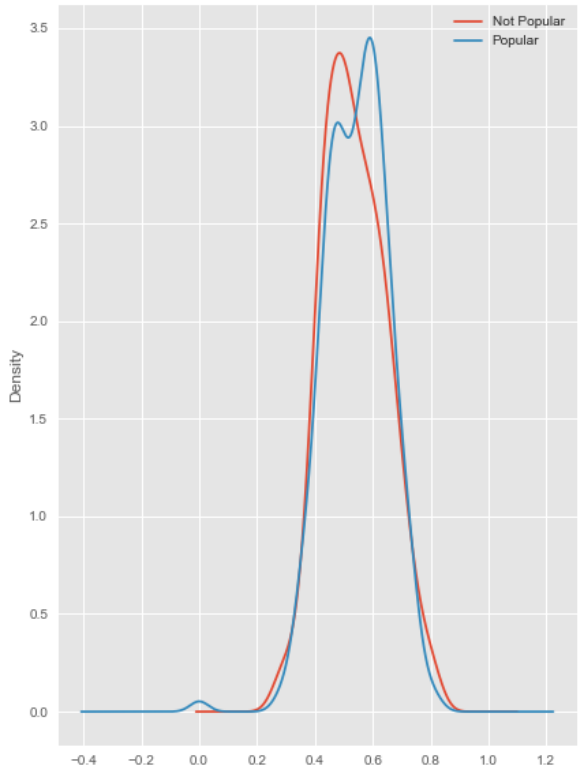
```
C:\Users\jsble\Anaconda3\lib\site-packages\ipykernel_launcher.py:14: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#in
dexing-view-versus-copy
```

In [189]:
```
# now lets see the statistics of these attributes
from pandas.tools.plotting import boxplot

# group the original data and the support vectors
df_grouped_support = df_support.groupby(['popularity'])
df_grouped = df_imputed.groupby(['popularity'])

# plot KDE of Different variables
vars_to_plot = ['n_tokens_content', 'n_unique_tokens', 'n_non_stop_unique_tokens', 'num_hrefs',
'num_self_hrefs', 'kw_min_min', 'kw_max_avg', 'kw_avg_avg', 'weekend', 'global_rate_positive_word
s', 'global_rate_negative_words', 'rate_negative_words']

for v in vars_to_plot:
    plt.figure(figsize=(15,10))
    #plt.figure(figsize=(10,4))
    # plot support vector stats
    plt.subplot(1,2,1)
    ax = df_grouped_support[v].plot.kde()
    plt.legend(['Not Popular','Popular'])
    plt.title(v+' (Instances chosen as Support Vectors)')

    # plot original distributions
    plt.subplot(1,2,2)
    ax = df_grouped[v].plot.kde()
    plt.legend(['Not Popular','Popular'])
    plt.title(v+' (Original)')
```

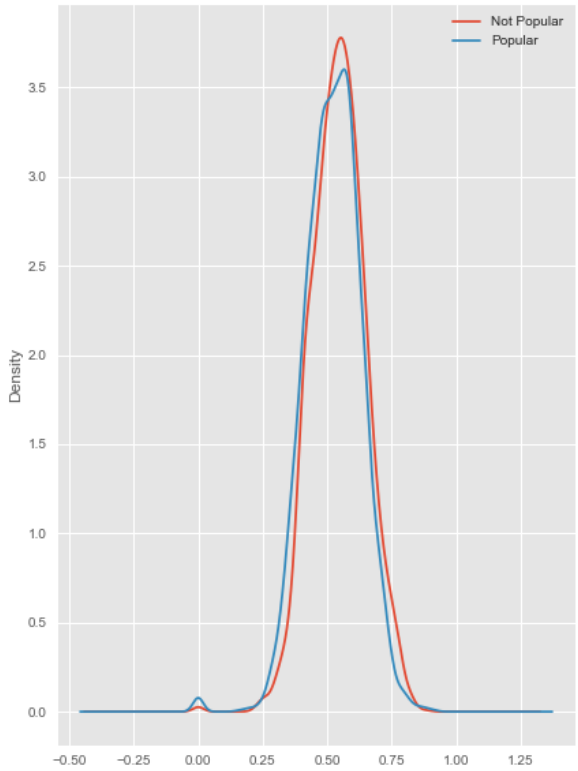




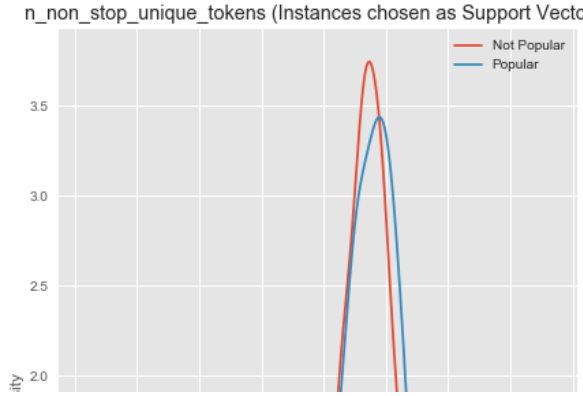
n_unique_tokens (Instances chosen as Support Vectors)



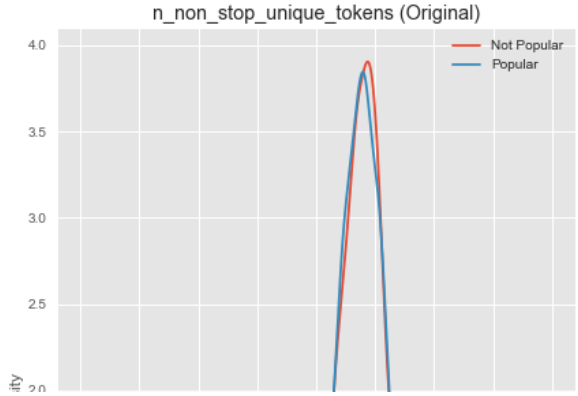
n_unique_tokens (Original)

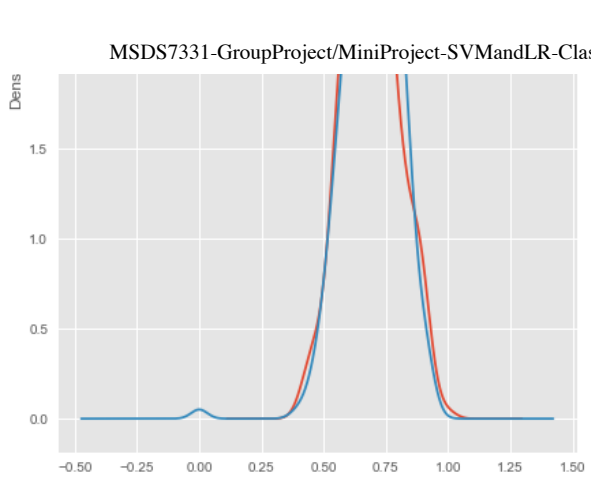


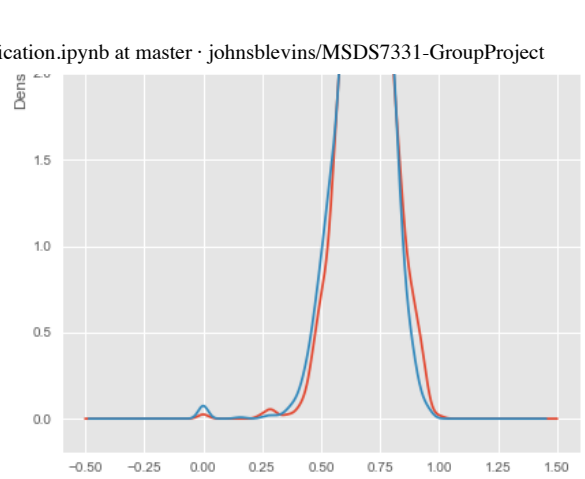
n_non_stop_unique_tokens (Instances chosen as Support Vectors)

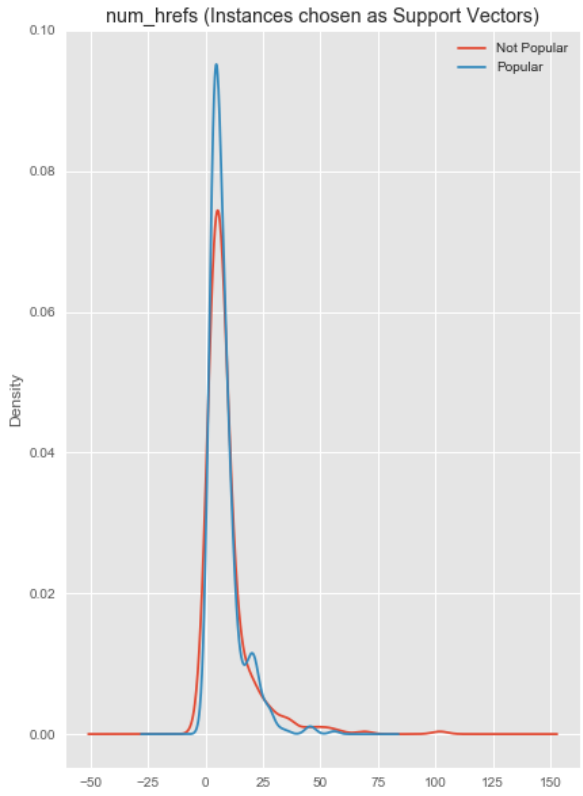


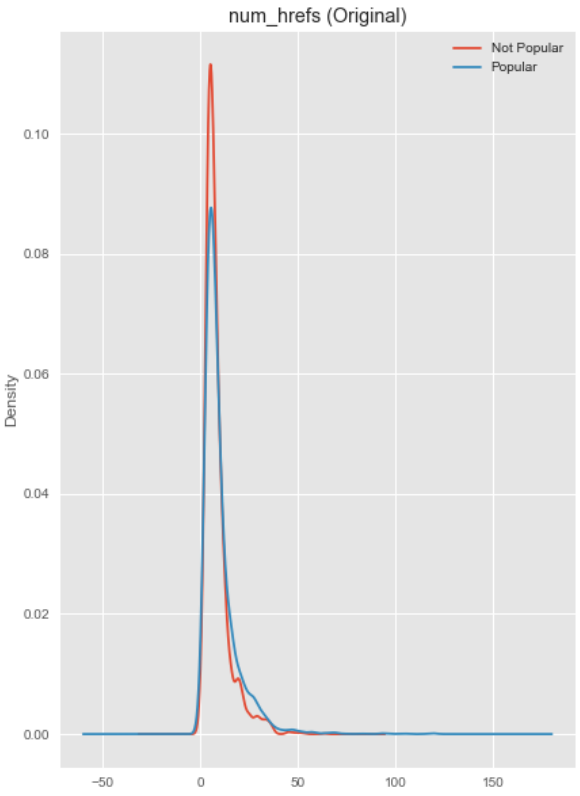
n_non_stop_unique_tokens (Original)

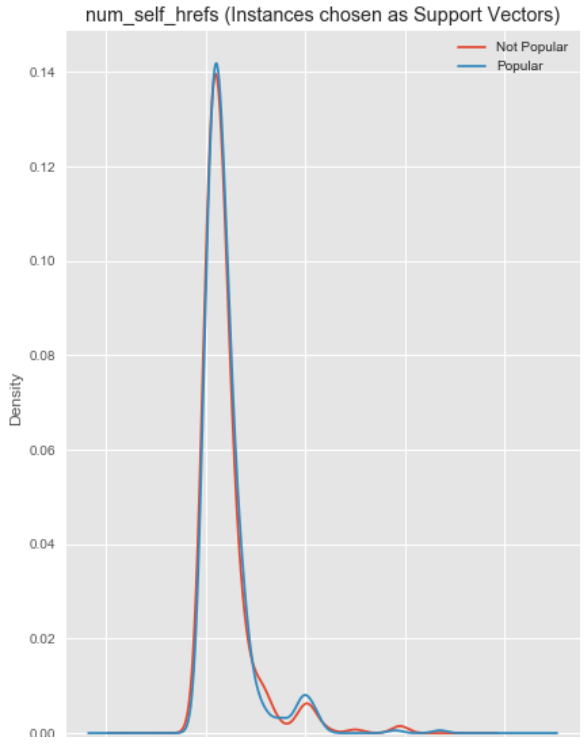


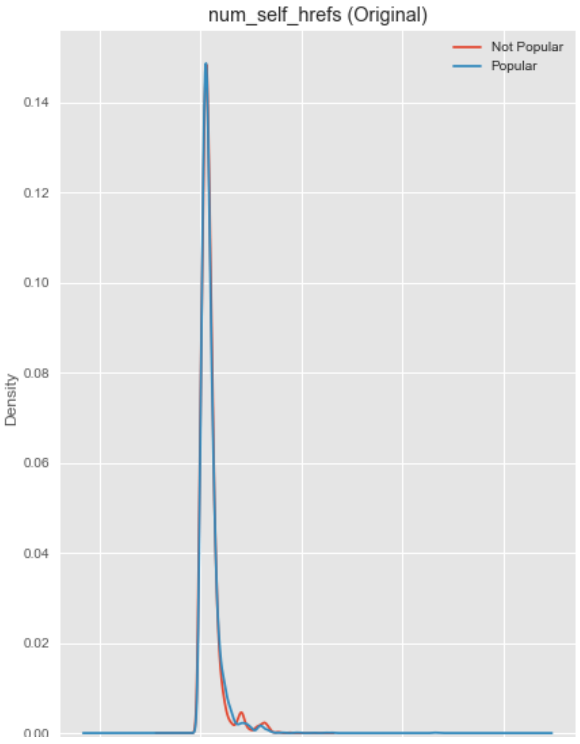

Dens
1.5
1.0
0.5
0.0
−0.50 −0.25 0.00 0.25 0.50 0.75 1.00 1.25 1.50

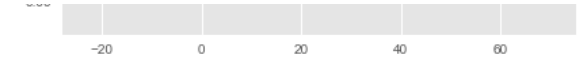

Dens
1.5
1.0
0.5
0.0
−0.50 −0.25 0.00 0.25 0.50 0.75 1.00 1.25 1.50

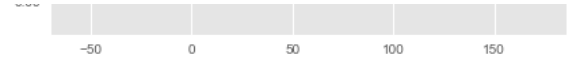

num_hrefs (Instances chosen as Support Vectors)
Not Popular
Popular
Density
0.10
0.08
0.06
0.04
0.02
0.00
−50 −25 0 25 50 75 100 125 150

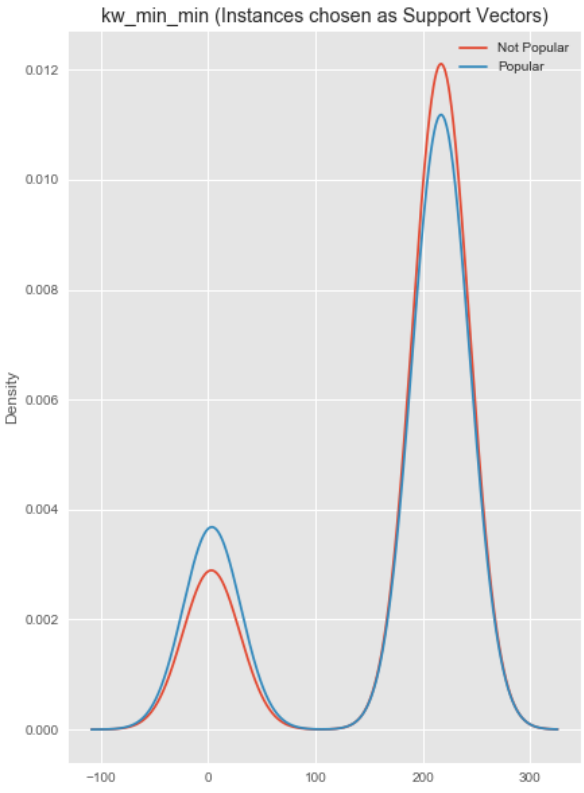

num_hrefs (Original)
Not Popular
Popular
Density
0.10
0.08
0.06
0.04
0.02
0.00
−50 0 50 100 150

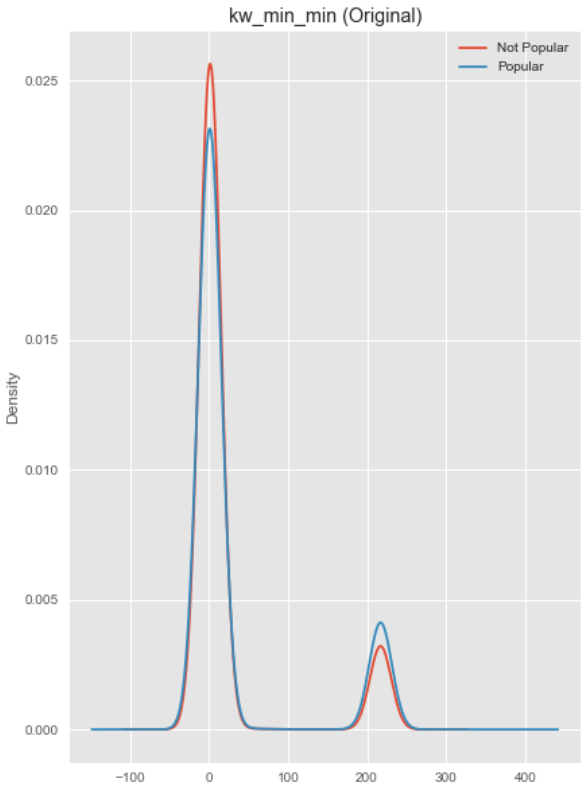

num_self_hrefs (Instances chosen as Support Vectors)
Not Popular
Popular
Density
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00

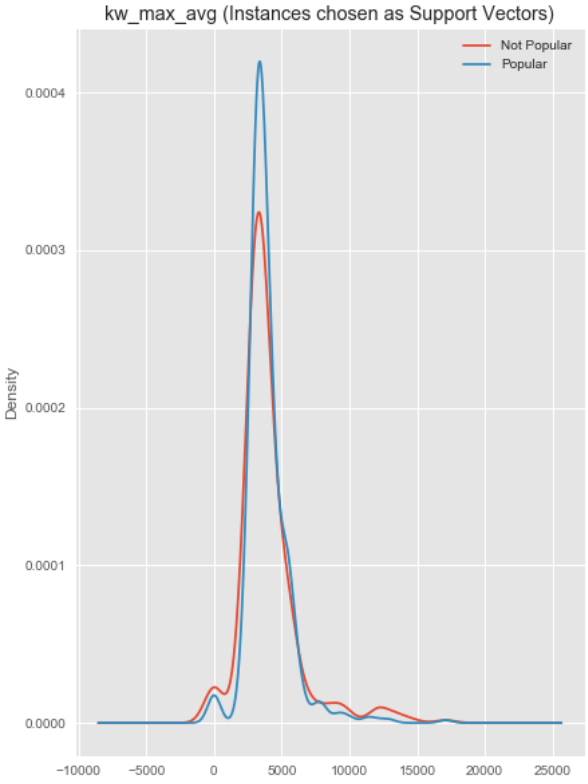

num_self_hrefs (Original)
Not Popular
Popular
Density
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00

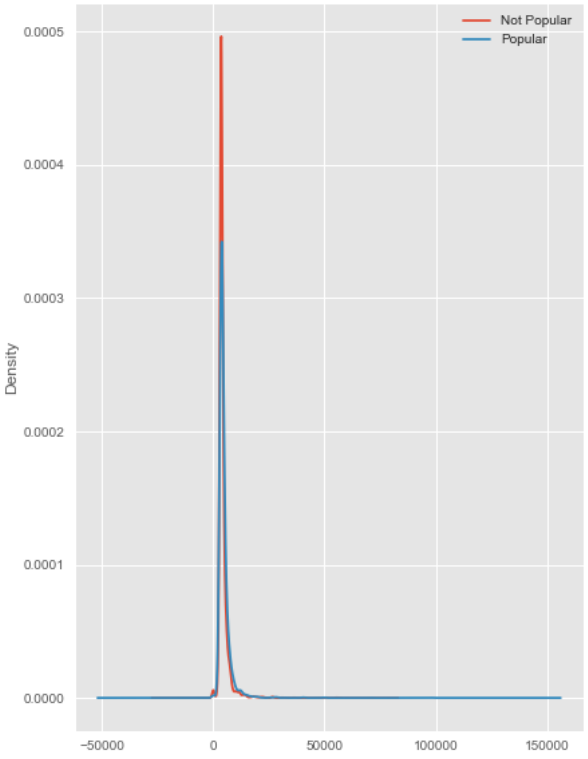

−20
0
20
40
60

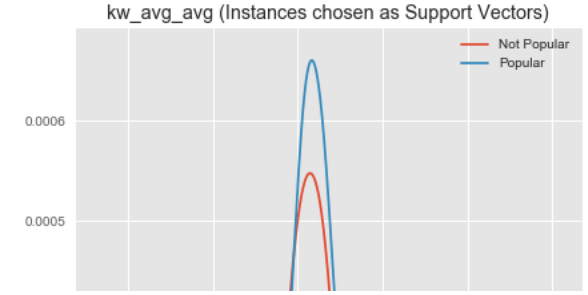

−50
0
50
100
150

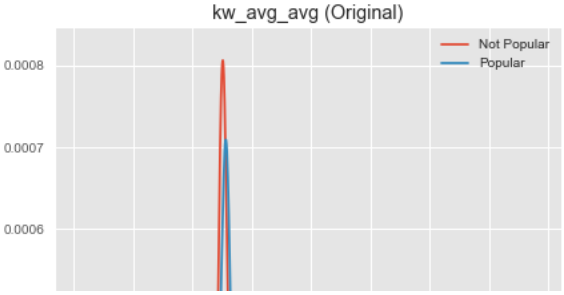

kw_min_min (Instances chosen as Support Vectors)
Not Popular
Popular
Density
0.012
0.010
0.008
0.006
0.004
0.002
0.000
−100
0
100
200
300

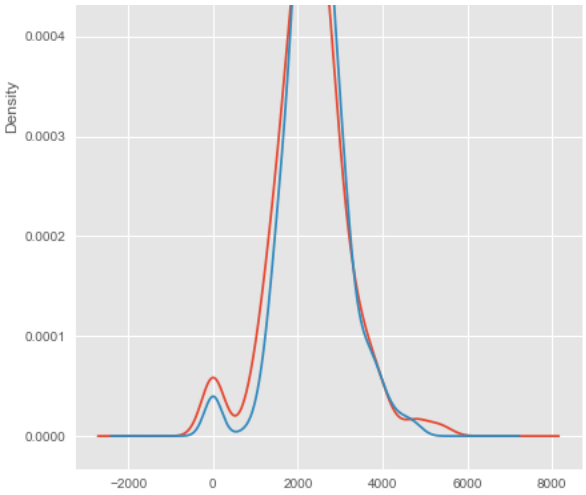

kw_min_min (Original)
Not Popular
Popular
Density
0.025
0.020
0.015
0.010
0.005
0.000
−100
0
100
200
300
400

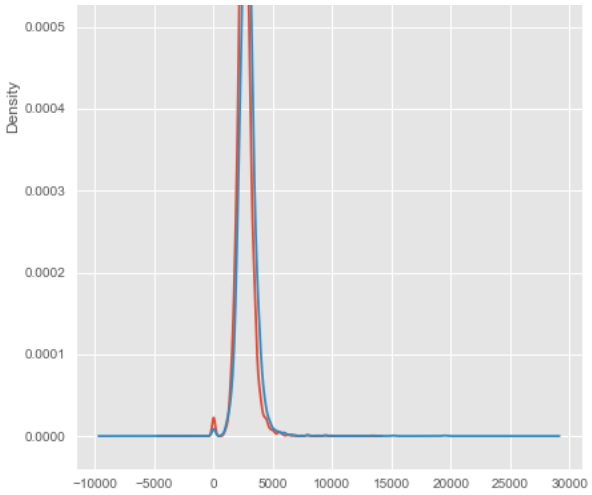

kw_max_avg (Instances chosen as Support Vectors)
Not Popular
Popular
Density
0.0004
0.0003
0.0002
0.0001
0.0000
−10000
−5000
0
5000
10000
15000
20000
25000

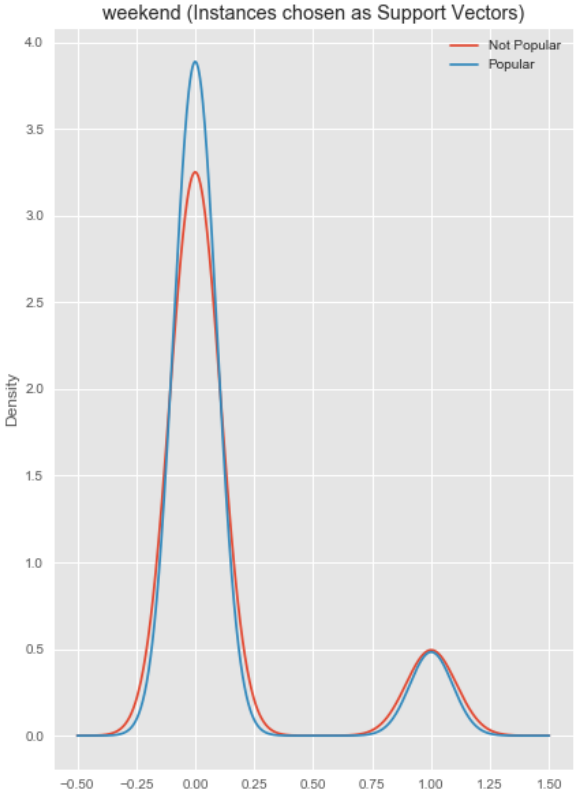

kw_max_avg (Original)
Not Popular
Popular
Density
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
−50000
0
50000
100000
150000

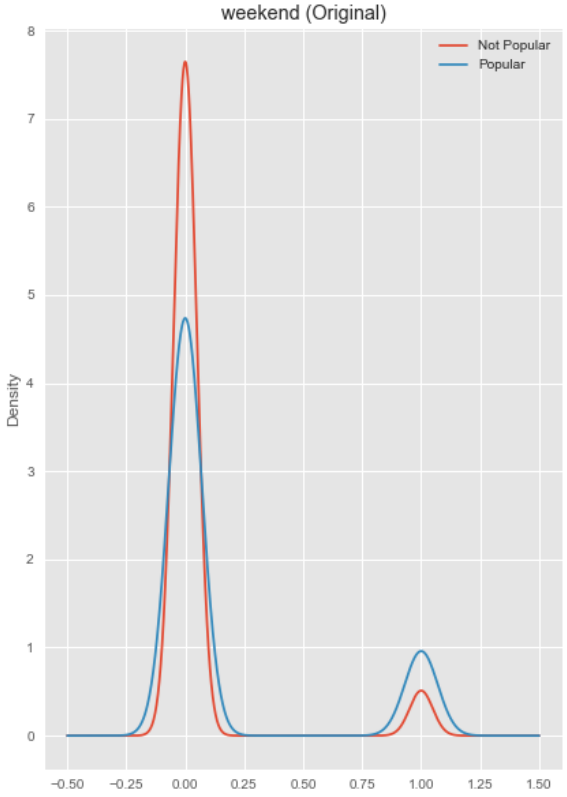

kw_avg_avg (Instances chosen as Support Vectors)
Not Popular
Popular
0.0006
0.0005

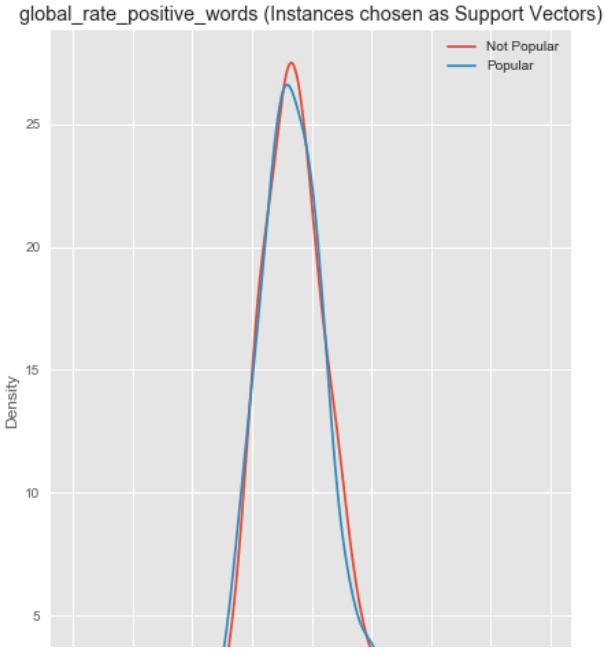

kw_avg_avg (Original)
Not Popular
Popular
0.0008
0.0007
0.0006

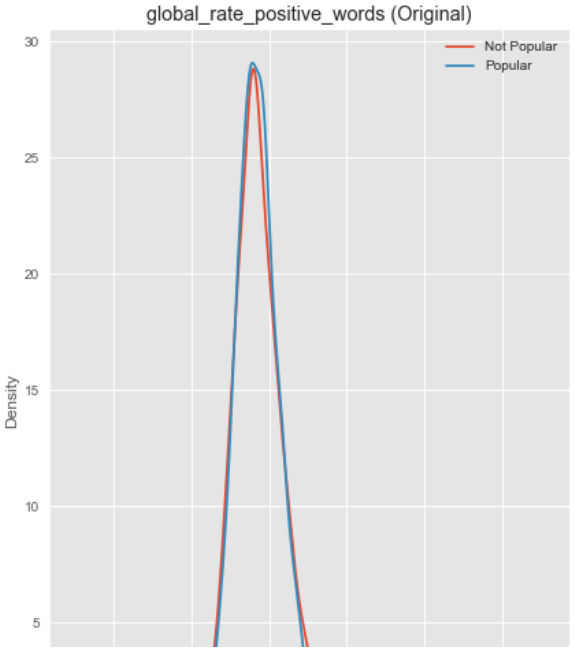

Density
0.0004
0.0003
0.0002
0.0001
0.0000
−2000
0
2000
4000
6000
8000

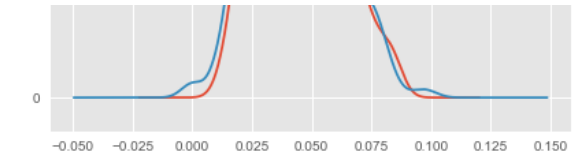

Density
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
−10000
−5000
0
5000
10000
15000
20000
25000
30000

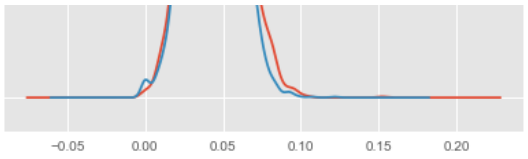

weekend (Instances chosen as Support Vectors)
Not Popular
Popular
Density
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50

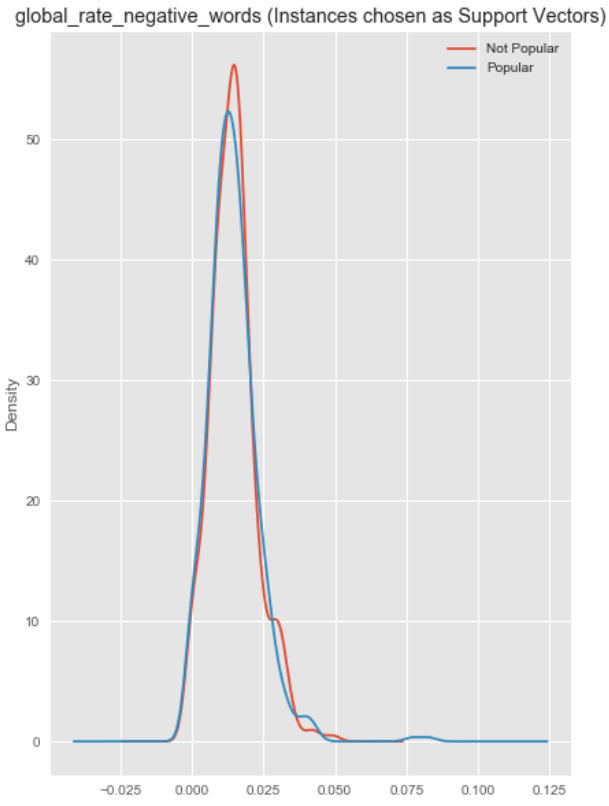

weekend (Original)
Not Popular
Popular
Density
8
7
6
5
4
3
2
1
0
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50

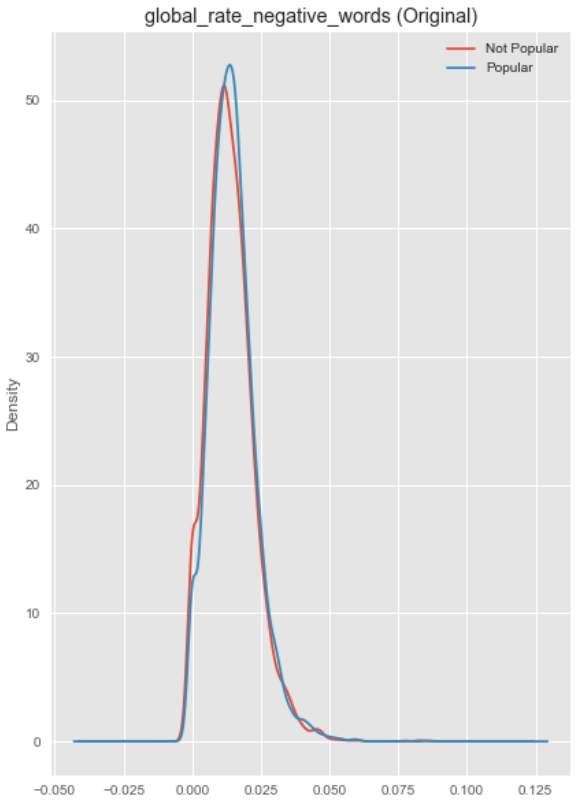

global_rate_positive_words (Instances chosen as Support Vectors)
Not Popular
Popular
Density
25
20
15
10
5

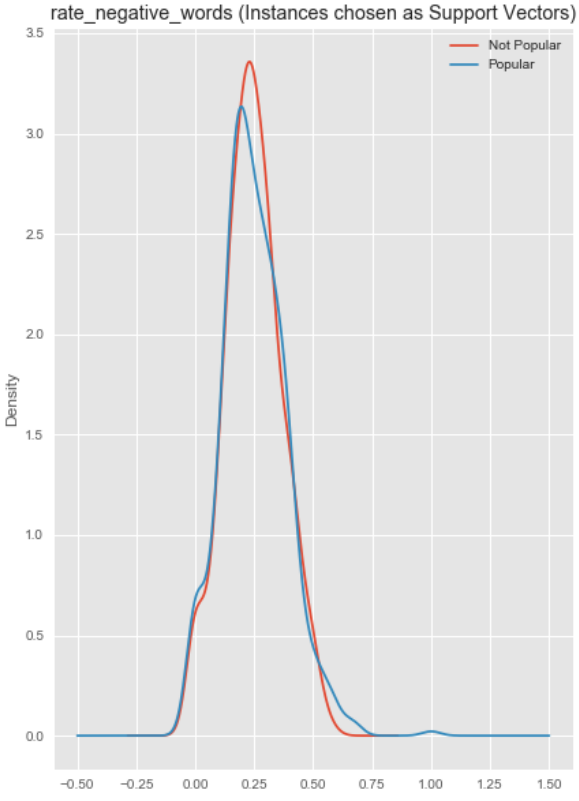

global_rate_positive_words (Original)
Not Popular
Popular
Density
30
25
20
15
10
5

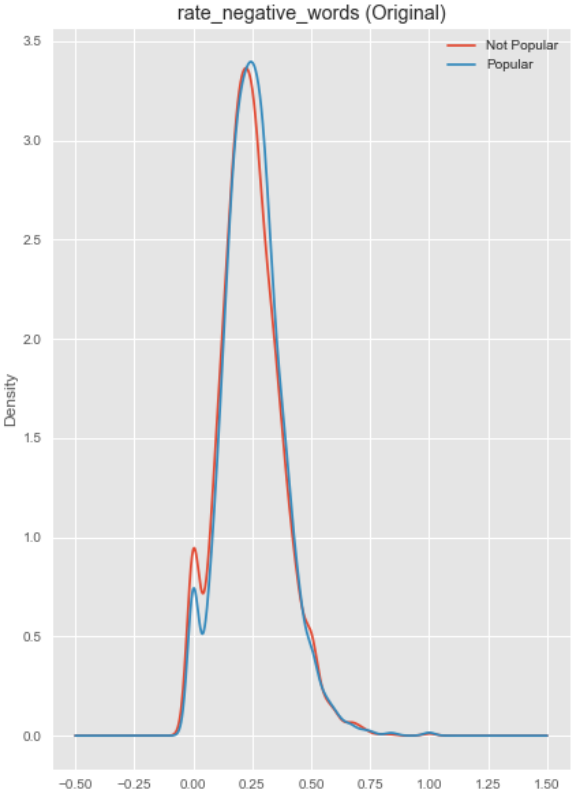












When looking at the density graphs we can see which attributes support vectors are more or less popular than the original. For example we can see in n_tokens_content that the instances chosen as support vectors for popular instances were about the same as the original data. However, there were less non popular instances.

In terms of coefficients with larger weights, the instances chosen as support vectors for n_unique_tokens was less dense than the original data,

but roughly the same distribution. This is also the same for n_non_stop_unique_tokens. For kw_avg_avg, this is also true but also there are more popular instances chosen in the support vector than non popular. The opposite is true in the original data. This is also the case for kw_max_avg.

We did not have to use stochastic gradient descent on our data and therefore did not have to take any subsamples of the data.

Advantages/Disadvantages of Each Model

The support vector model offers slightly better performance accuracy than the logistric regression model. On average the accuracy is around 1% better for SVM with the dataset under analysis. This is expected since SVM is typically expected to perform only marginally better than logistic regression with a smaller number of dimensions. Because logistic regression depends much more on probability functions versus the use of predictor attributes to determine a response variable this is sufficient. For an analysis which has hundreds or even thousands of explanatory veriables SVM would likely provide significantly more accuracy. In terms of training time the support vector model does take significantly longer to complete. On average it takes around 34 times longer to train the SVM model. This is expected due to the calculation times invovled with calcualting the hyperplanes on multiple dimensions. Logistic regression simply tries to determine a probability for a logistic function of a linear combination of the attributes which takes much fewer iterations.

Sources

https://www.quora.com/Machine-Learning/How-does-one-decide-on-which-kernel-to-choose-for-an-SVM-RBF-vs-linear-vs-poly-kernel (https://www.quora.com/Machine-Learning/How-does-one-decide-on-which-kernel-to-choose-for-an-SVM-RBF-vs-linear-vs-poly-kernel)