2. Post_Clean_LogisticRegression

August 16, 2018

1 Logistic Regression on Amazon Reviews Dataset (Part II)

1.1 Data Source:

The preprocessing step has produced final.sqlite file after doing the data preparation & cleaning. The review text is now devoid of punctuations, HTML markups and stop words.

1.2 Objective:

To find optimal lambda using GridSearchCV & RandomSearchCV on standardized feature vectors obtained from BoW, tf-idf, W2V and tf-idf weighted W2V featurizations. To study the impact on sparsity upon increasing lambda.

Find Precision, Recall, F1 Score, Confusion Matrix, Accuracy of 10-fold cross validation with GridSearch and RandomSearch with optimal Logistic Regression regression model on vectorized input data, for BoW, tf-idf, W2V and tf-idf weighted W2V featurizations. TPR, TNR, FPR and FNR is calculated for all.

After finding the optimal model, **do Perturbation test** to remove multicollinear features. **Find top n words** using the weight vector, w.

1.3 At a glance:

Random Sampling is done to reduce input data size and time based slicing to split into training and testing data. **The optimal lambda is found out using GridSearchCV & RandomSearchCV with a range of lamda values to search (for GridSearch) and an uniform distribution (for RandomSearchCV.**

The Precision, Recall, F1 Score, Confusion Matrix, Accuracy metrics are found out for all 4 featurizations. A normal distribution noise is added for perturbnatino test and the identified multicollinear features are removed. Then the top 'n' words are found out after removal of multicollinear features based on highest values of `|w|`.

2 Preprocessed Data Loading

```
In [17]: #loading libraries for LR
         import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         from sklearn.cross_validation import train_test_split
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cross_validation import cross_val_score
from collections import Counter
from sklearn.metrics import accuracy_score
from sklearn import cross_validation

#loading libraries for scikit learn, nlp, db, plot and matrix.
import sqlite3
import pdb
import pandas as pd
import numpy as np
import nltk
import string
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.metrics import roc_curve, auc
from nltk.stem.porter import PorterStemmer


# using the SQLite Table to read data.
con = sqlite3.connect('./final.sqlite')

#filtering only positive and negative reviews i.e.
# not taking into consideration those reviews with Score=3
final = pd.read_sql_query("""
SELECT *
FROM Reviews
""", con)

print(final.head(3))
print(final.shape)
```

```
    index      Id   ProductId          UserId            ProfileName  \
0  138706  150524  0006641040   ACITT7DI6IDDL        shari zychinski
1  138688  150506  0006641040  A2IW4PEEKO2ROU                  Tracy
2  138689  150507  0006641040  A1S4A3IQ2MU7V4  sally sue "sally sue"

   HelpfulnessNumerator  HelpfulnessDenominator     Score        Time  \
0                     0                       0  positive   939340800
1                     1                       1  positive  1194739200
2                     1                       1  positive  1191456000
```

```
                                   Summary  \
0                   EVERY book is educational
1   Love the book, miss the hard cover version
2               chicken soup with rice months

                                              Text  \
0  this witty little book makes my son laugh at l...
1  I grew up reading these Sendak books, and watc...
2  This is a fun way for children to learn their ...

                                       CleanedText
0  b'witti littl book make son laugh loud recit c...
1  b'grew read sendak book watch realli rosi movi...
2  b'fun way children learn month year learn poem...
(364171, 12)
```

3 Random Sampling & Time Based Slicing

In [18]:
```python
# To randomly sample the data and sort based on time before doing train/ test split.
# The slicing into train & test data is done thereafter.

num_points = 20000

# used to format headings
bold = '\033[1m'
end = '\033[0m'

# you can use random_state for reproducibility
sampled_final = final.sample(n=num_points, random_state=2)


#Sorting data according to Time in ascending order
sorted_final = sampled_final.sort_values('Time', axis=0,
                ascending=True, inplace=False, kind='quicksort', na_position='last')

# fetching the outcome class
y = sorted_final['Score']

def class2num(response):
    if (response == 'positive'):
        return 1
    else:
        return 0

y_bin = list(map(class2num, y))
```

```
X_train, X_test, y_train, y_test = train_test_split(
            sorted_final, y_bin, test_size=0.3, random_state=42)
```

4 Custom Defined Functions

5 user defined functions are written to

```
a) Perform GridSearchCV & RandomSearchCV for Optimal Alpha Estimation.

b) Compute Logistic Regression Classifier Performance Metrics.

c) Find Most Frequent Words.

d) Analyze Sparsity for increasing Lambda.

e) Perturbation Test with a Normal Distributed Noise.
```

4.1 a) GridSearchCV & RandomSearchCV for Optimal Alpha Estimation

```
In [19]: # source: https://chrisalbon.com/machine_learning/
         # model_selection/hyperparameter_tuning_using_random_search/
         # some parts of the below code are from the above link.

         # Cross Validation using RandomizedSearchCV & GridSearchCV

         import numpy
         import math
         from scipy.stats import uniform
         import matplotlib.pyplot as plt
         from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import RandomizedSearchCV
         from sklearn.linear_model import LogisticRegression

         def gridRandomCV(X_train_vect, X_test_vect, c_max, title_cf=''):

             # empty list that will hold cv scores
             cv_scores = []

             # Create regularization penalty space
             penalty = ['l1', 'l2']

             # Create regularization hyperparameter distribution using uniform distribution
             # This distribution is constant between loc and loc + scale.
             C = uniform(loc=0, scale=c_max)

             # Create hyperparameter options
             hyperparameters = dict(C=C, penalty=penalty)
```

```
##################################################################
# Cross Validation using RandomizedSearchCV
# Create randomized search 10-fold cross validation and 100 iterations
model = RandomizedSearchCV(LogisticRegression(), hyperparameters,
                random_state=1, n_iter=100, cv=10, verbose=0, n_jobs=-1)

# Fit randomized search
best_model = model.fit(X_train_vect, y_train)

best_regularizer = best_model.best_estimator_.get_params()['penalty']

# View best hyperparameters
print(bold + '\nBest Penalty:', best_regularizer)

optimal_lambda_rcv = best_model.best_estimator_.get_params()['C']
print('RandomizedSearchCV: Best C:', optimal_lambda_rcv, end, '\n')

means = best_model.cv_results_['mean_test_score']
stds = best_model.cv_results_['std_test_score']
print ("Mean Test Score (+/-) Standard Deviation for Parameters: ")
for mean, std, params in zip(
        means, stds, best_model.cv_results_['params']):
    print("%0.3f (+/-%0.03f) for %r"
          % (mean, std * 2, params))

print('\nThe optimal value of lambda using RandomizedSearchCV is %f.'
                                                  % (1/optimal_lambda_rcv))

compute_metrics(best_model, X_test_vect,
                    title_cf="Confusion Matrix: RandomizedSearchCV")
##################################################################


##################################################################
# Cross Validation using GridSearchCV
inv_lambda_values = [10**-5, 10**-4, 10**-3, 10**-2, 10**-1,
                       10**0, 10**1, 10**2, 10**3, 10**4, 10**5]

tuned_parameters = [{'C': inv_lambda_values}, {'penalty': penalty}]

model = GridSearchCV(LogisticRegression(),
                     tuned_parameters, scoring = 'f1', cv=10)
model.fit(X_train_vect, y_train)

means = model.cv_results_['mean_test_score']
#     stds = model.cv_results_['std_test_score']
#     for mean, std, params in zip(means, stds, model.cv_results_['params']):
```

```
#         print("%0.3f (+/-%0.03f) for %r"
#               % (mean, std * 2, params))
#     print(type(model.cv_results_['params']))
#     print(model.cv_results_['params'])

    # determining best lambda
    optimal_lambda_gcv = model.cv_results_['params'][means.argmax()].get('C')
    print('\nGridSearchCV: Best C:', optimal_lambda_gcv)
    print(
        '\nThe optimal value of lambda using GridSearchCV is %f.'
                                                    % (1/optimal_lambda_gcv))

    compute_metrics(model, X_test_vect, title_cf="Confusion Matrix: GridSearchCV")
    ###################################################################

    return optimal_lambda_rcv, best_regularizer
```

4.2 b) Compute Logistic Regression Classifier Performance Metrics

```
In [20]: # ========================== LR with alpha = optimal_alpha ==========================
         #To compute the performance metrics of Logistic Regression classifier

         import seaborn as sn
         from sklearn.metrics import *

         def compute_metrics(logR_optimal, X_test_vect, title_cf="Confusion Matrix"):

             # predict the response
             pred = logR_optimal.predict(X_test_vect)

             print(bold + '\n\nMetric Analysis of Logistic Classifier for Optimal Lamdba' + end

             # evaluate accuracy
             acc = accuracy_score(y_test, pred) * 100
             print('\nAccuracy \t= %f' % acc)

             precision = precision_score(y_test, pred) * 100
             print('Precision \t= %f' % precision)

             recall = recall_score(y_test, pred) * 100
             print('Recall \t\t= %f' % recall)

             f1score = f1_score(y_test, pred) * 100
             print('F1 Score \t= %f' % f1score)

             confusion = confusion_matrix(y_test, pred)
             print(bold + "\n\nConfusion Matrix" + end)
```

```
    plt.figure()
    plt.title(title_cf)
    df_cm = pd.DataFrame(confusion, range(2), range(2))
    sn.set(font_scale=1.4)#for label size
    sn.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt="d")# font size

    (tn, fp, fn, tp) = confusion.ravel()
    print("\nTrue Negatives = " + str(tn))
    print("True Positives = " + str(tp))
    print("False Negatives = " + str(fn))
    print("False Positives = " + str(fp))

    actual_positives = tp+fn
    actual_negatives = tn+fp
    print("\nTotal Actual Positives = " + str(actual_positives))
    print("Total Actual Negatives = " + str(actual_negatives))

    print("\nTrue Positive Rate(TPR) = " + str(round(tp/actual_positives, 2)))
    print("True Negative Rate(TNR) = " + str(round(tn/actual_negatives, 2)))
    print("False Positive Rate(FPR) = " + str(round(fp/actual_negatives, 2)))
    print("False Negative Rate(FNR) = " + str(round(fn/actual_positives, 2)))
```

4.3 c) Find Most Frequent Words

In [21]:
```
# To find out the out top words based on absolute values of w
# Exclusion of collinear features done using mask

from itertools import compress

def find_top_words(vect, weights, mask, nwords):

    # Sort the absolute value of weights
    weight_sorted = abs(weights).argsort()

    # Exclude the collinear features
    features = vect.get_feature_names()
    features_masked = list(compress(features, list(~mask)))

    # find top words
    top_words = np.take(features_masked,
                        weight_sorted[weight_sorted.size-nwords:])

    print(bold + "\n\nTop Words: "+ end)
    for id, word in enumerate(top_words):
        print("\t" + word + "\t\t Weight: " + str(
            round(weights[weight_sorted[weight_sorted.size-nwords+id]], 2)))
```

4.4 d) Analyze Sparsity for increasing Lambda

In [22]:
```
# More Sparsity (i.e. fewer elements of W* being non-zero)
# by increasing Lambda (decreasing C)

def testL1_increaseLambda(X_train_vect, X_test_vect):

    # empty list that will hold values
    lamdas = []
    sparsities = []
    f1scores = []

    invlamda = 1000000

    print(bold +
        '\n\nSparsity Analysis of L1 Regularizer for increasing Lambda' + end)

    # iterate to reach lowest value of invlamda
    while invlamda > 10**-2:

        clf = LogisticRegression(C=invlamda, penalty='l1')
        clf.fit(X_train_vect, y_train)
        w = clf.coef_

        pred = clf.predict(X_test_vect)
        f1score = f1_score(y_test, pred) * 100


        lamda = round(1/invlamda, 6)
        sparsity = round(np.count_nonzero(w))
        f1score = round(f1score, 2)

        lamdas.append(math.log(lamda, 10))
        sparsities.append(sparsity)
        f1scores.append(f1score)

        print(bold +"\nSparsity vs Performance: Lambda = "
                                              + str(lamda) + end)
        print("Sparsity =" + str(sparsity))
        print("F1 Score =" + str(f1score))

        invlamda *= 10**-1

    plt.figure()
    plt.plot(lamdas, sparsities)
    plt.xlabel('Log (Lambda)')
    plt.ylabel('# of Non-Zero Elements')
    plt.title('Increasing Lambda: Sparsity Plot')
```

```
plt.figure()
plt.plot(lamdas, f1scores)
plt.xlabel('Log (Lambda)')
plt.ylabel('F1 Score')
plt.title('Increasing Lambda: F1 Score Plot')
```

4.5 e) Perturbation Test with a Normal Distributed Noise

Sparsity of input vector is preserved for BoW and tf-idf featurizations. For W2V and tf-idf W2V the features are dense.

```
In [23]: # Perturbation Test after adding N(0, 0.01)

         def doPertubationTest(X_train_vect, invLambda, regularizer, isSparse):

             clf = LogisticRegression(C=invLambda, penalty = regularizer)
             clf.fit(X_train_vect, y_train)
             w = clf.coef_
             w = w[0]
             print("\nLength of Weight Vector (Before Removing Collinearity): "
                                                                  + str(len(w)))

             # Generate epsilon = normal distribution with mean = 0 and std = 0.01
             epsilon = np.random.normal(loc=0.0, scale=0.01,size = X_train_vect.shape)

             # To add epsilon only to non-zero elements
             mask = X_train_vect != 0

             #if sparse matrix from bow or tfidf then convert to dense array
             if (isSparse):
                 mask = mask.toarray()

             X_train_vect[mask] = (X_train_vect[mask].astype(float) +
                                   epsilon[mask].astype(float)).astype(float)

             # To calculate weight vector, w, after perturbation
             clf.fit(X_train_vect, y_train)
             w_pert = clf.coef_
             w_pert = w_pert[0]


             # To find the % change in weights per feature
             w_change = w_pert/w

             dist = numpy.linalg.norm(w-w_pert)
             print("Distance between Weight vectors before & after Perturbation  = "
                                                          + str(round(dist,2)))
```

```
        # if the percent change > threshold then that feature is multicollinear
        percent_change = 0.05

        # Eliminate collinear features and return weight vector to find top features.
        mask = (w_change > 1+percent_change) | (w_change < 1-percent_change)

        print("Multicollinear Features = " + str((w_change[mask]).size))

        return w[~mask], mask
```

5 BoW

BoW will result in a **sparse matrix with huge number of features** as it creates a feature for each unique word in the review.

For Binary BoW feature representation, CountVectorizer is declared as float, as the values can take non-integer values on further processing. Top n words are found out after checking for multicollinearity.

```
In [24]: # BoW Featurisation, Standardisation, Grid Search and Random Search,
         # Impact of Sparsity on increasing lambda, Perturbation test to remove
         # multicollinear features, Find top n words using weight vector.

         # from sklearn.decomposition import TruncatedSVD
         from sklearn.random_projection import sparse_random_matrix
         from sklearn.preprocessing import StandardScaler

         #BoW
         count_vect = CountVectorizer(dtype="float") #in scikit-learn
         X_train_vect = count_vect.fit_transform(X_train['CleanedText'].values)
         X_train_vect.get_shape()

         #BoW Test
         X_test_vect = count_vect.transform(X_test['CleanedText'].values)

         # Standardisation. Set "with_mean=False" to preserve sparsity
         scaler = StandardScaler(copy=False, with_mean=False).fit(X_train_vect)
         X_train_vect = scaler.transform(X_train_vect)
         scaler = StandardScaler(copy=False, with_mean=False).fit(X_test_vect)
         X_test_vect = scaler.transform(X_test_vect)

         print(bold + "\n\n1) Grid Search and Random Search CV using Logistic Regression"+ end)

         # Do both grid Search and Random Search.
         # The function returns optimal value of lambda
         # sets the maximum value of C to be 4 for RandomCV
         optimal_lambda, best_regularizer = gridRandomCV(X_train_vect, X_test_vect, 4)
```

```python
        # Do pertubation test to check multicollinearity.
        # Get weight vector after removing collinear features.
        weights_non_collinear, mask = doPertubationTest(
                                X_train_vect, optimal_lambda, best_regularizer, True)

        print("\nLength of Weight Vector (After Removing Collinearity): "
                                    + str(len(weights_non_collinear)))

        # To print top n=20 words
        find_top_words(count_vect, weights_non_collinear, mask, 20)
```

```
1) Grid Search and Random Search CV using Logistic Regression
Best Penalty: l1RandomizedSearchCV: Best C: 0.1782075141790469

Mean Test Score (+/-) Standard Deviation for Parameters:
0.887 (+/-0.018) for {'C': 1.668088018810296, 'penalty': 'l1'}
0.870 (+/-0.017) for {'C': 3.730229437354635, 'penalty': 'l2'}
0.875 (+/-0.016) for {'C': 1.209330290527359, 'penalty': 'l2'}
0.876 (+/-0.016) for {'C': 0.9443559078079042, 'penalty': 'l2'}
0.877 (+/-0.016) for {'C': 0.7450408455106836, 'penalty': 'l2'}
0.871 (+/-0.016) for {'C': 2.67898414721392, 'penalty': 'l2'}
0.884 (+/-0.018) for {'C': 2.155266936013428, 'penalty': 'l1'}
0.888 (+/-0.018) for {'C': 1.2530940677291005, 'penalty': 'l1'}
0.891 (+/-0.019) for {'C': 0.8178089989260697, 'penalty': 'l1'}
0.891 (+/-0.019) for {'C': 0.9183088549193021, 'penalty': 'l1'}
0.871 (+/-0.016) for {'C': 2.681870040713609, 'penalty': 'l2'}
0.872 (+/-0.016) for {'C': 1.828819231947953, 'penalty': 'l2'}
0.893 (+/-0.015) for {'C': 0.5615477543809351, 'penalty': 'l1'}
0.870 (+/-0.017) for {'C': 3.113556945346134, 'penalty': 'l2'}
0.870 (+/-0.017) for {'C': 3.87304630287759, 'penalty': 'l2'}
0.879 (+/-0.016) for {'C': 0.3712032345629517, 'penalty': 'l2'}
0.882 (+/-0.017) for {'C': 3.5055566091841532, 'penalty': 'l1'}
0.882 (+/-0.016) for {'C': 3.3165876294685663, 'penalty': 'l1'}
0.882 (+/-0.017) for {'C': 0.15621913293152945, 'penalty': 'l2'}
0.898 (+/-0.012) for {'C': 0.23697280520625386, 'penalty': 'l1'}
0.879 (+/-0.017) for {'C': 0.3933873353322004, 'penalty': 'l2'}
0.871 (+/-0.017) for {'C': 2.6866163896885373, 'penalty': 'l2'}
0.884 (+/-0.018) for {'C': 2.1326611398920683, 'penalty': 'l1'}
0.875 (+/-0.016) for {'C': 1.1585185621832497, 'penalty': 'l2'}
0.883 (+/-0.018) for {'C': 2.7460037107263346, 'penalty': 'l1'}
0.873 (+/-0.016) for {'C': 1.6501553660121044, 'penalty': 'l2'}
0.871 (+/-0.017) for {'C': 3.00057725977987, 'penalty': 'l2'}
0.871 (+/-0.017) for {'C': 2.6425429209520117, 'penalty': 'l2'}
0.875 (+/-0.016) for {'C': 1.1217759682576207, 'penalty': 'l2'}
0.877 (+/-0.016) for {'C': 0.8884981901414992, 'penalty': 'l2'}
0.873 (+/-0.016) for {'C': 1.7915741047036207, 'penalty': 'l2'}
```

```
0.897 (+/-0.014) for {'C': 0.3846890417818467, 'penalty': 'l1'}
0.889 (+/-0.019) for {'C': 1.151101354345395, 'penalty': 'l1'}
0.872 (+/-0.017) for {'C': 2.099197507481782, 'penalty': 'l2'}
0.883 (+/-0.018) for {'C': 2.715342131759564, 'penalty': 'l1'}
0.882 (+/-0.017) for {'C': 3.641793527172406, 'penalty': 'l1'}
0.872 (+/-0.017) for {'C': 1.9662926371213532, 'penalty': 'l2'}
0.872 (+/-0.017) for {'C': 2.2636481098567725, 'penalty': 'l2'}
0.892 (+/-0.016) for {'C': 0.5869142996232406, 'penalty': 'l1'}
0.890 (+/-0.018) for {'C': 1.0439159184622273, 'penalty': 'l1'}
0.879 (+/-0.016) for {'C': 0.40933771531130336, 'penalty': 'l2'}
0.870 (+/-0.016) for {'C': 3.7997525541305586, 'penalty': 'l2'}
0.886 (+/-0.018) for {'C': 1.6567170781076106, 'penalty': 'l1'}
0.882 (+/-0.017) for {'C': 3.0619404177634837, 'penalty': 'l1'}
0.883 (+/-0.017) for {'C': 2.6551785808791553, 'penalty': 'l1'}
0.871 (+/-0.016) for {'C': 3.1696143431840764, 'penalty': 'l2'}
0.872 (+/-0.017) for {'C': 2.3462201620079717, 'penalty': 'l2'}
0.872 (+/-0.017) for {'C': 2.1631527584038586, 'penalty': 'l2'}
0.893 (+/-0.016) for {'C': 0.5571053890030342, 'penalty': 'l1'}
0.874 (+/-0.017) for {'C': 1.5103373646043785, 'penalty': 'l2'}
0.892 (+/-0.018) for {'C': 0.6614167884677311, 'penalty': 'l1'}
0.888 (+/-0.019) for {'C': 1.445044080846484, 'penalty': 'l1'}
0.871 (+/-0.016) for {'C': 3.003248412544622, 'penalty': 'l2'}
0.894 (+/-0.016) for {'C': 0.5052595401795532, 'penalty': 'l1'}
0.871 (+/-0.017) for {'C': 2.4946888282224355, 'penalty': 'l2'}
0.888 (+/-0.019) for {'C': 1.4157563551932077, 'penalty': 'l1'}
0.889 (+/-0.019) for {'C': 1.0797115670601043, 'penalty': 'l1'}
0.883 (+/-0.016) for {'C': 3.011153412702417, 'penalty': 'l1'}
0.881 (+/-0.016) for {'C': 3.8593601885935422, 'penalty': 'l1'}
0.885 (+/-0.019) for {'C': 1.9924362785129461, 'penalty': 'l1'}
0.879 (+/-0.015) for {'C': 0.45898389181350074, 'penalty': 'l2'}
0.898 (+/-0.013) for {'C': 0.2562693193528691, 'penalty': 'l1'}
0.884 (+/-0.018) for {'C': 2.313558457548527, 'penalty': 'l1'}
0.887 (+/-0.019) for {'C': 1.519213147074788, 'penalty': 'l1'}
0.881 (+/-0.017) for {'C': 3.613518082249015, 'penalty': 'l1'}
0.875 (+/-0.017) for {'C': 1.2054419816395736, 'penalty': 'l2'}
0.884 (+/-0.017) for {'C': 2.4685796544828955, 'penalty': 'l1'}
0.884 (+/-0.018) for {'C': 2.3145357229428916, 'penalty': 'l1'}
0.870 (+/-0.017) for {'C': 3.543768397243098, 'penalty': 'l2'}
0.872 (+/-0.017) for {'C': 1.882563230649684, 'penalty': 'l2'}
0.883 (+/-0.018) for {'C': 2.493440463167211, 'penalty': 'l1'}
0.890 (+/-0.019) for {'C': 1.076004205031312, 'penalty': 'l1'}
0.871 (+/-0.017) for {'C': 2.763587670067696, 'penalty': 'l2'}
0.880 (+/-0.016) for {'C': 0.28032598413051657, 'penalty': 'l2'}
0.893 (+/-0.016) for {'C': 0.548542998515511, 'penalty': 'l1'}
0.892 (+/-0.019) for {'C': 0.7678243129497218, 'penalty': 'l1'}
0.880 (+/-0.015) for {'C': 0.26400069088824996, 'penalty': 'l2'}
0.872 (+/-0.017) for {'C': 2.032990150269156, 'penalty': 'l2'}
0.870 (+/-0.017) for {'C': 3.6920981421859334, 'penalty': 'l2'}
```

```
0.872 (+/-0.017) for {'C': 2.075447503455608, 'penalty': 'l2'}
0.885 (+/-0.016) for {'C': 0.07952053535918235, 'penalty': 'l2'}
0.896 (+/-0.014) for {'C': 0.4294612118097092, 'penalty': 'l1'}
0.876 (+/-0.016) for {'C': 0.9848442704121836, 'penalty': 'l2'}
0.875 (+/-0.016) for {'C': 1.1548713308224396, 'penalty': 'l2'}
0.871 (+/-0.018) for {'C': 2.2112879147430635, 'penalty': 'l2'}
0.889 (+/-0.019) for {'C': 1.1132753528408204, 'penalty': 'l1'}
0.875 (+/-0.017) for {'C': 1.116734716044558, 'penalty': 'l2'}
0.884 (+/-0.018) for {'C': 2.280266641664378, 'penalty': 'l1'}
0.872 (+/-0.017) for {'C': 2.24412087702284, 'penalty': 'l2'}
0.870 (+/-0.016) for {'C': 3.2519798815438947, 'penalty': 'l2'}
0.876 (+/-0.016) for {'C': 0.9318970953640817, 'penalty': 'l2'}
0.871 (+/-0.017) for {'C': 2.4517923709112597, 'penalty': 'l2'}
0.882 (+/-0.017) for {'C': 3.4541674182377147, 'penalty': 'l1'}
0.873 (+/-0.017) for {'C': 1.7112505303692886, 'penalty': 'l2'}
0.893 (+/-0.016) for {'C': 0.5458209026427401, 'penalty': 'l1'}
0.892 (+/-0.017) for {'C': 0.6133539116782978, 'penalty': 'l1'}
0.900 (+/-0.013) for {'C': 0.1782075141790469, 'penalty': 'l1'}
0.870 (+/-0.017) for {'C': 3.1278338499578635, 'penalty': 'l2'}
0.871 (+/-0.017) for {'C': 2.8519559215307067, 'penalty': 'l2'}
0.871 (+/-0.017) for {'C': 2.6172950867580167, 'penalty': 'l2'}

The optimal value of lambda using RandomizedSearchCV is 5.611436.
```

Metric Analysis of Logistic Classifier for Optimal Lamdba

```
Accuracy         = 90.016667
Precision         = 92.818740
Recall                 = 95.552481
F1 Score         = 94.165774
```

Confusion Matrix

```
True Negatives = 567
True Positives = 4834
False Negatives = 225
False Positives = 374

Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.96
True Negative Rate(TNR) = 0.6
False Positive Rate(FPR) = 0.4
False Negative Rate(FNR) = 0.04

GridSearchCV: Best C: 0.001

The optimal value of lambda using GridSearchCV is 1000.000000.
```

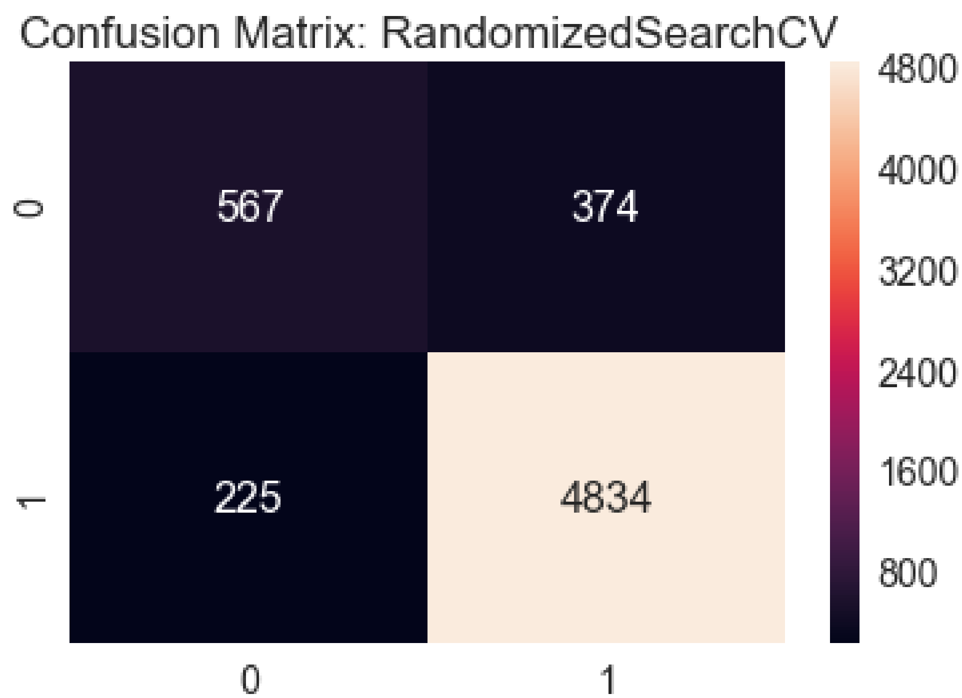
Metric Analysis of Logistic Classifier for Optimal Lamdba

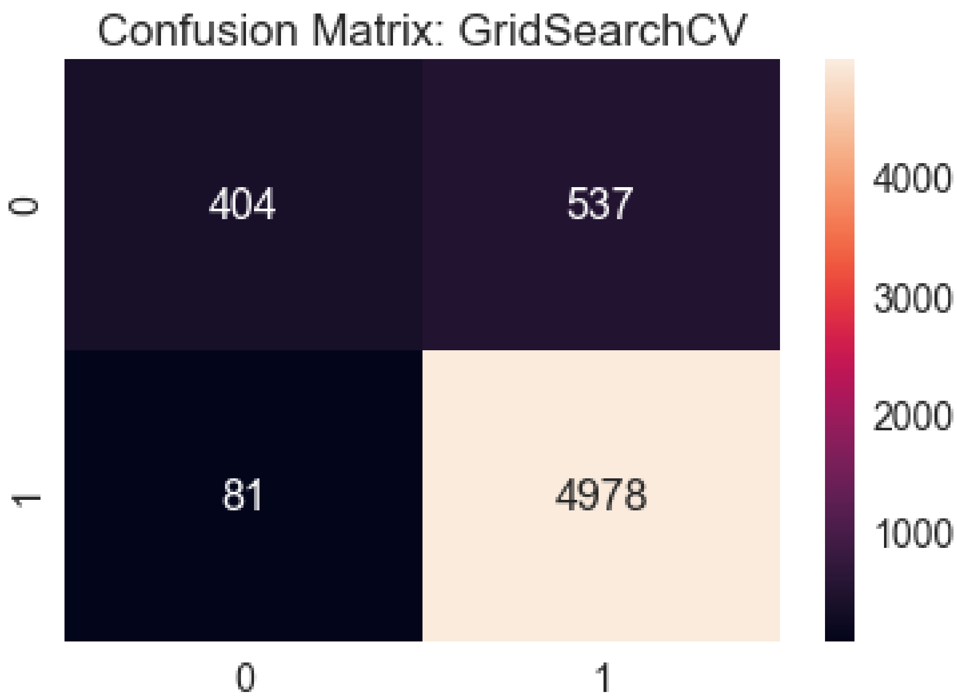
```
Accuracy         = 89.700000
Precision         = 90.262919
Recall                = 98.398893
F1 Score         = 94.155476
```
Confusion Matrix

```
True Negatives = 404
True Positives = 4978
False Negatives = 81
False Positives = 537

Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.98
True Negative Rate(TNR) = 0.43
False Positive Rate(FPR) = 0.57
False Negative Rate(FNR) = 0.02

Length of Weight Vector (Before Removing Collinearity): 15114


C:\Anaconda\lib\site-packages\ipykernel_launcher.py:32: RuntimeWarning: divide by zero encounte
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:32: RuntimeWarning: invalid value encounter
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:42: RuntimeWarning: invalid value encounter
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:42: RuntimeWarning: invalid value encounter


Distance between Weight vectors before & after Perturbation  = 0.3
Multicollinear Features = 829

Length of Weight Vector (After Removing Collinearity): 14285
```
Top Words:

```
        bit                  Weight: 0.35
        year                  Weight: 0.36
        enjoy                  Weight: 0.37
        favorit                  Weight: 0.4
        tasti                  Weight: 0.41
        product                  Weight: -0.43
        nice                  Weight: 0.47
        tast                  Weight: -0.47
        excel                  Weight: 0.48
        amaz                  Weight: 0.48
        easi                  Weight: 0.48
        return                  Weight: -0.48
        day                  Weight: 0.5
        disappoint                  Weight: -0.5
```

```
delici                          Weight: 0.61
perfect                           Weight: 0.68
good                          Weight: 0.69
best                          Weight: 0.87
love                          Weight: 0.95
great                           Weight: 1.19
```





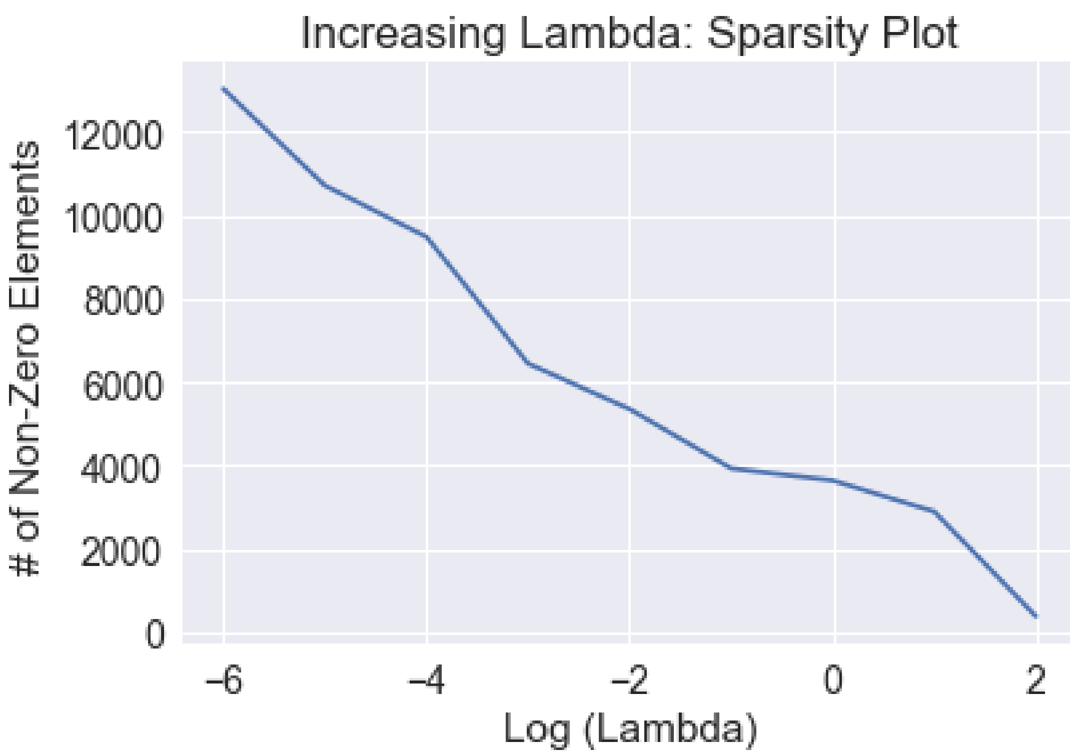
6 Sparsity vs F1 score Plot

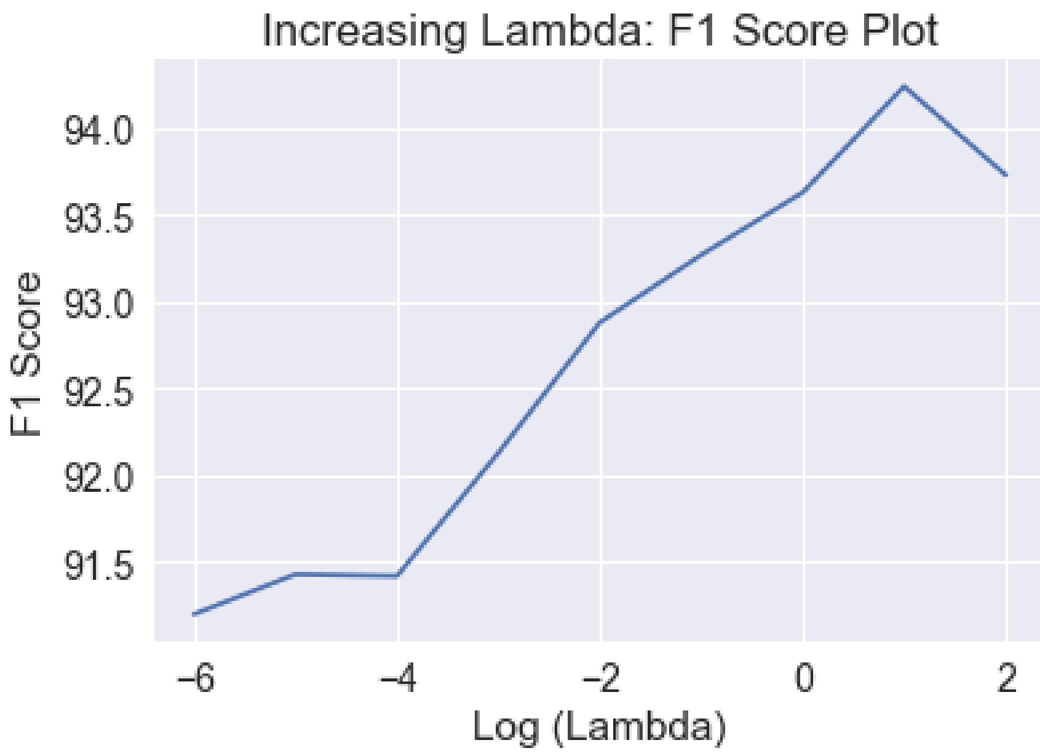
The variation of sparsity corresponding to varying values of lambda is plotted and the lambda with the highest accuracy is identified. The optimal model can be found out using the sparsity vs f1 score plot also.

```
In [25]: # To study the variation of sparsity vs f1 score for increasing values of lambda.
         # here the train/ test vector is based on BoW featurization.
         testL1_increaseLambda(X_train_vect, X_test_vect)

         # Here Sparsity = # of non-zero elements.
         # it is found that the number of zero elements increases as lambda is increased.
```

```
Sparsity Analysis of L1 Regularizer for increasing Lambda
Sparsity vs Performance: Lambda = 1e-06
Sparsity =13034
F1 Score =91.2
Sparsity vs Performance: Lambda = 1e-05
Sparsity =10716
F1 Score =91.43
Sparsity vs Performance: Lambda = 0.0001
Sparsity =9482
F1 Score =91.42
Sparsity vs Performance: Lambda = 0.001
```

```
Sparsity =6441
F1 Score =92.13
```
Sparsity vs Performance: Lambda = 0.01
```
Sparsity =5353
F1 Score =92.88
```
Sparsity vs Performance: Lambda = 0.1
```
Sparsity =3926
F1 Score =93.27
```
Sparsity vs Performance: Lambda = 1.0
```
Sparsity =3637
F1 Score =93.63
```
Sparsity vs Performance: Lambda = 10.0
```
Sparsity =2891
F1 Score =94.24
```
Sparsity vs Performance: Lambda = 100.0
```
Sparsity =376
F1 Score =93.73
```





7 tf-IDF

Sparse matrix generated from tf-IDF is fed in to GridSearch and RandomSearch Logistic Regression Cross Validator to find the optimal lambda value. Performance metrics of optimal LR with tf-idf featurization is found.

In [26]:

```
# TFID Featurisation, Standardisation, Grid Search and Random Search,
# Perturbation test to remove multicollinear features, Find top n words.

from sklearn.random_projection import sparse_random_matrix
from sklearn.preprocessing import StandardScaler

# TFID
count_vect = TfidfVectorizer(dtype="float") #in scikit-learn
X_train_vect = count_vect.fit_transform(X_train['CleanedText'].values)
X_train_vect.get_shape()

# TFID Test
X_test_vect = count_vect.transform(X_test['CleanedText'].values)

# Standardisation. Set "with_mean=False" to preserve sparsity
```

```python
scaler = StandardScaler(copy=False, with_mean=False).fit(X_train_vect)
X_train_vect = scaler.transform(X_train_vect)
scaler = StandardScaler(copy=False, with_mean=False).fit(X_test_vect)
X_test_vect = scaler.transform(X_test_vect)

print(bold + "\n\n1) Grid Search and Random Search CV using Logistic Regression"+ end)

# Do both grid Search and Random Search.
# The function returns optimal value of lambda
# sets the maximum value of C to be 10**4 for RandomCV
optimal_lambda, best_regularizer = gridRandomCV(X_train_vect, X_test_vect, 10**4)

# To check sparsity and f1 score for increasing values of lambda
# testL1_increaseLambda(X_train_vect, X_test_vect)

# Do pertubation test to check multicollinearity.
# Get weight vector after removing collinear features.
weights_non_collinear, mask = doPertubationTest(
                                X_train_vect, optimal_lambda, best_regularizer, True)

print("\nLength of Weight Vector (After Removing Collinearity): "
                                    + str(len(weights_non_collinear)))

# To print top n words
find_top_words(count_vect, weights_non_collinear, mask, 20)
```

```
1) Grid Search and Random Search CV using Logistic Regression
Best Penalty: l1RandomizedSearchCV: Best C: 445.51878544761723

Mean Test Score (+/-) Standard Deviation for Parameters:
0.852 (+/-0.013) for {'C': 4170.22004702574, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 9325.573593386587, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 3023.3257263183978, 'penalty': 'l2'}
0.852 (+/-0.018) for {'C': 2360.8897695197606, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 1862.602113776709, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 6697.4603680348, 'penalty': 'l2'}
0.849 (+/-0.013) for {'C': 5388.167340033569, 'penalty': 'l1'}
0.853 (+/-0.014) for {'C': 3132.735169322751, 'penalty': 'l1'}
0.854 (+/-0.010) for {'C': 2044.5224973151744, 'penalty': 'l1'}
0.852 (+/-0.015) for {'C': 2295.7721372982555, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 6704.675101784022, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 4572.048079869883, 'penalty': 'l2'}
0.855 (+/-0.013) for {'C': 1403.8693859523378, 'penalty': 'l1'}
0.851 (+/-0.017) for {'C': 7783.892363365335, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 9682.615757193975, 'penalty': 'l2'}
0.853 (+/-0.018) for {'C': 928.0080864073792, 'penalty': 'l2'}
0.850 (+/-0.013) for {'C': 8763.891522960383, 'penalty': 'l1'}
0.849 (+/-0.015) for {'C': 8291.469073671416, 'penalty': 'l1'}
```

```
0.854 (+/-0.018) for {'C': 390.54783232882363, 'penalty': 'l2'}
0.859 (+/-0.013) for {'C': 592.4320130156347, 'penalty': 'l1'}
0.853 (+/-0.018) for {'C': 983.468338330501, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 6716.540974221343, 'penalty': 'l2'}
0.850 (+/-0.014) for {'C': 5331.652849730171, 'penalty': 'l1'}
0.852 (+/-0.018) for {'C': 2896.2964054581244, 'penalty': 'l2'}
0.854 (+/-0.011) for {'C': 6865.0092768158365, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 4125.388415030261, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 7501.443149449675, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 6606.357302380029, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 2804.4399206440517, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 2221.245475353748, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 4478.935261759052, 'penalty': 'l2'}
0.854 (+/-0.014) for {'C': 961.7226044546168, 'penalty': 'l1'}
0.854 (+/-0.018) for {'C': 2877.7533858634874, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 5247.993768704456, 'penalty': 'l2'}
0.849 (+/-0.015) for {'C': 6788.35532939891, 'penalty': 'l1'}
0.852 (+/-0.018) for {'C': 9104.483817931015, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 4915.731592803383, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 5659.120274641931, 'penalty': 'l2'}
0.853 (+/-0.010) for {'C': 1467.2857490581016, 'penalty': 'l1'}
0.852 (+/-0.012) for {'C': 2609.789796155568, 'penalty': 'l1'}
0.853 (+/-0.018) for {'C': 1023.3442882782584, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 9499.381385326396, 'penalty': 'l2'}
0.851 (+/-0.014) for {'C': 4141.792695269027, 'penalty': 'l1'}
0.850 (+/-0.016) for {'C': 7654.85104440871, 'penalty': 'l1'}
0.850 (+/-0.019) for {'C': 6637.946452197888, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 7924.035857960191, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 5865.550405019929, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 5407.881896009646, 'penalty': 'l2'}
0.857 (+/-0.015) for {'C': 1392.7634725075854, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 3775.843411510946, 'penalty': 'l2'}
0.856 (+/-0.010) for {'C': 1653.5419711693278, 'penalty': 'l1'}
0.853 (+/-0.011) for {'C': 3612.61020211621, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 7508.121031361556, 'penalty': 'l2'}
0.853 (+/-0.013) for {'C': 1263.148850448883, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 6236.7220705560885, 'penalty': 'l2'}
0.849 (+/-0.012) for {'C': 3539.390887983019, 'penalty': 'l1'}
0.850 (+/-0.016) for {'C': 2699.2789176502606, 'penalty': 'l1'}
0.847 (+/-0.014) for {'C': 7527.883531756042, 'penalty': 'l1'}
0.848 (+/-0.020) for {'C': 9648.400471483856, 'penalty': 'l1'}
0.849 (+/-0.014) for {'C': 4981.090696282366, 'penalty': 'l1'}
0.852 (+/-0.018) for {'C': 1147.459729533752, 'penalty': 'l2'}
0.857 (+/-0.013) for {'C': 640.6732983821728, 'penalty': 'l1'}
0.850 (+/-0.012) for {'C': 5783.896143871318, 'penalty': 'l1'}
0.850 (+/-0.012) for {'C': 3798.0328676869703, 'penalty': 'l1'}
0.851 (+/-0.016) for {'C': 9033.795205622539, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 3013.604954098934, 'penalty': 'l2'}
```

```
0.850 (+/-0.014) for {'C': 6171.449136207239, 'penalty': 'l1'}
0.850 (+/-0.015) for {'C': 5786.339307357229, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 8859.420993107746, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 4706.40807662421, 'penalty': 'l2'}
0.850 (+/-0.016) for {'C': 6233.601157918028, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 2690.01051257828, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 6908.96917516924, 'penalty': 'l2'}
0.853 (+/-0.019) for {'C': 700.8149603262914, 'penalty': 'l2'}
0.854 (+/-0.018) for {'C': 1371.3574962887776, 'penalty': 'l1'}
0.853 (+/-0.015) for {'C': 1919.5607823743044, 'penalty': 'l1'}
0.853 (+/-0.018) for {'C': 660.0017272206248, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 5082.475375672891, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 9230.245355464833, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 5188.61875863902, 'penalty': 'l2'}
0.855 (+/-0.018) for {'C': 198.80133839795587, 'penalty': 'l2'}
0.854 (+/-0.013) for {'C': 1073.653029524273, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 2462.1106760304588, 'penalty': 'l2'}
0.852 (+/-0.018) for {'C': 2887.178327056099, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 5528.219786857659, 'penalty': 'l2'}
0.852 (+/-0.010) for {'C': 2783.188382102051, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 2791.8367901113947, 'penalty': 'l2'}
0.850 (+/-0.020) for {'C': 5700.666604160945, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 5610.302192557099, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 8129.9497038597365, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 2329.7427384102043, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 6129.4809272781495, 'penalty': 'l2'}
0.851 (+/-0.015) for {'C': 8635.418545594286, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 4278.126325923222, 'penalty': 'l2'}
0.856 (+/-0.012) for {'C': 1364.5522566068503, 'penalty': 'l1'}
0.852 (+/-0.012) for {'C': 1533.3847791957444, 'penalty': 'l1'}
0.859 (+/-0.017) for {'C': 445.51878544761723, 'penalty': 'l1'}
0.852 (+/-0.017) for {'C': 7819.584624894659, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 7129.889803826766, 'penalty': 'l2'}
0.852 (+/-0.017) for {'C': 6543.237716895042, 'penalty': 'l2'}

The optimal value of lambda using RandomizedSearchCV is 0.002245.
```

Metric Analysis of Logistic Classifier for Optimal Lamdba

```
Accuracy        = 86.033333
Precision        = 91.228756
Recall               = 92.310733
F1 Score        = 91.766555
```

Confusion Matrix

```
True Negatives = 492
True Positives = 4670
False Negatives = 389
False Positives = 449
```

```
Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.92
True Negative Rate(TNR) = 0.52
False Positive Rate(FPR) = 0.48
False Negative Rate(FNR) = 0.08

GridSearchCV: Best C: 0.001

The optimal value of lambda using GridSearchCV is 1000.000000.
```

Metric Analysis of Logistic Classifier for Optimal Lamdba

```
Accuracy         = 89.283333
Precision         = 90.000000
Recall                = 98.201226
F1 Score         = 93.921921
```

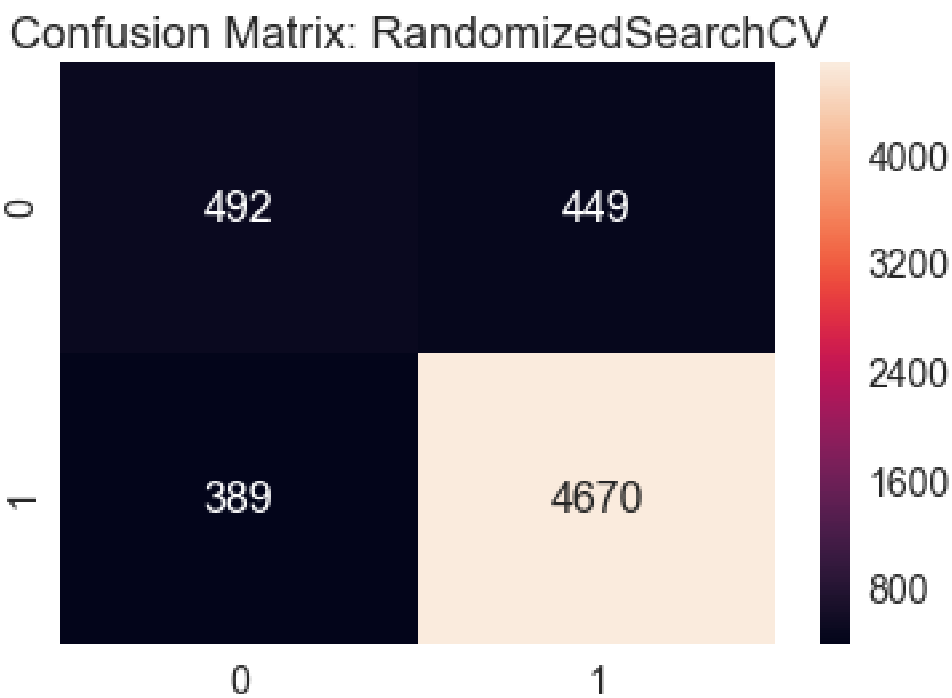
Confusion Matrix

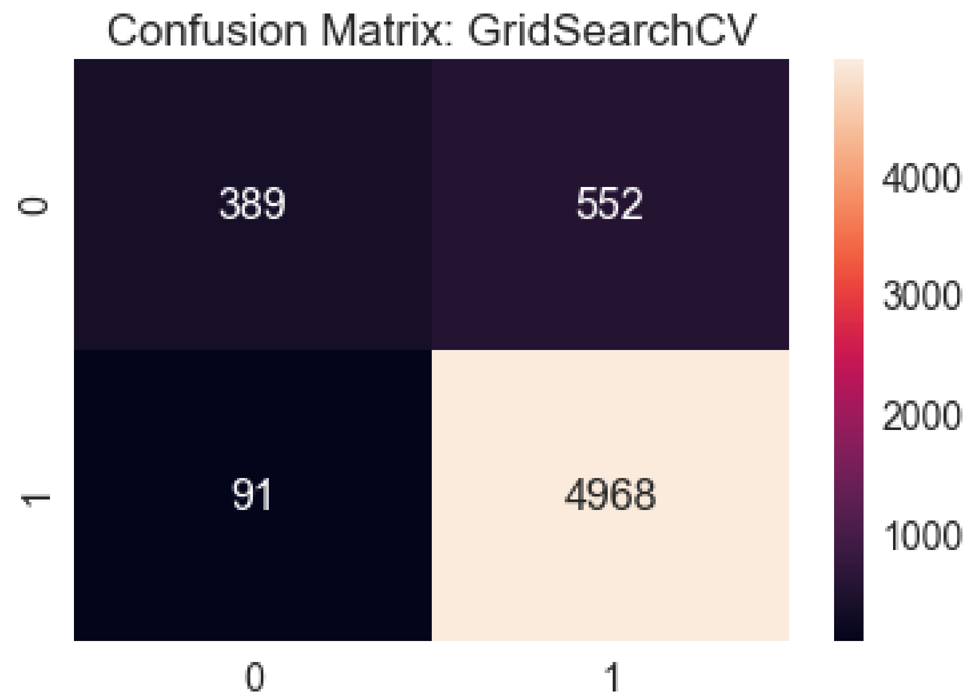
```
True Negatives = 389
True Positives = 4968
False Negatives = 91
False Positives = 552

Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.98
True Negative Rate(TNR) = 0.41
False Positive Rate(FPR) = 0.59
False Negative Rate(FNR) = 0.02

Length of Weight Vector (Before Removing Collinearity): 15114


C:\Anaconda\lib\site-packages\ipykernel_launcher.py:32: RuntimeWarning: divide by zero encounte
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:32: RuntimeWarning: invalid value encounter
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:42: RuntimeWarning: invalid value encounter
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:42: RuntimeWarning: invalid value encounter


Distance between Weight vectors before & after Perturbation  = 2.93
Multicollinear Features = 7557

Length of Weight Vector (After Removing Collinearity): 7557
```

Top Words:

```
        right                 Weight: 0.61
```

```
mellow                     Weight: 0.63
worst                     Weight: -0.63
delight                     Weight: 0.63
delic                     Weight: 0.63
definit                     Weight: 0.68
complaint                     Weight: 0.68
everyon                     Weight: 0.68
easier                     Weight: 0.73
often                     Weight: 0.74
unhealthi                     Weight: 0.75
threw                     Weight: -0.78
wonder                     Weight: 0.83
enjoy                     Weight: 0.95
perfect                     Weight: 1.05
excel                     Weight: 1.1
good                     Weight: 1.16
love                     Weight: 1.55
best                     Weight: 1.8
great                     Weight: 1.92
```





8 Word2Vec

Dense matrix generated from Word2Vec is fed in to GridSearch and RandomSearch Logistic Regression Cross Validator to find the optimal lambda value.

Performance metrics of optimal LR with W2V featurization is found. But we cannot find the top 'n' words when we use Word2Vec based featurization, because the feature doesnt correspond to a word in the vocabulary.

```
In [27]: # Train your own Word2Vec model using your own text corpus
         import gensim
         import re

         w2v_dim = 300

         def cleanhtml(sentence): #function to clean the word of any html-tags
             cleanr = re.compile('<.*?>')
             cleantext = re.sub(cleanr, ' ', sentence)
             return cleantext

         #function to clean the word of any punctuation or special characters
         def cleanpunc(sentence):
             cleaned = re.sub(r'[?|!|\'|"|#]',r'',sentence)
             cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned)
             return  cleaned
```

```
def trainW2V_model(reviewText):
    #select subset of points for fast execution
    i=0
    list_of_sent=[]

    for sent in reviewText:
        sent = str(sent, 'utf-8')
        filtered_sentence=[]
        sent=cleanhtml(sent)
        for w in sent.split():
            for cleaned_words in cleanpunc(w).split():
                if(cleaned_words.isalpha()):
                    filtered_sentence.append(cleaned_words.lower())
                else:
                    continue
        list_of_sent.append(filtered_sentence)

    w2v_model=gensim.models.Word2Vec(list_of_sent,min_count=5,size=w2v_dim, workers=4

    return w2v_model
```

```
C:\Anaconda\lib\site-packages\gensim\utils.py:1197: UserWarning: detected Windows; aliasing chu
  warnings.warn("detected Windows; aliasing chunkize to chunkize_serial")
```

In [28]:
```
# average Word2Vec
# compute average word2vec for each review.

def computeAvgW2V(w2vTrained_model, reviewText):
    sent_vectors = []; # the avg-w2v for each sentence/review is stored in this list

    for sent in reviewText: # for each review/sentence
        sent_vec = np.zeros(w2v_dim) # as word vectors are of zero length
        cnt_words =0; # num of words with a valid vector in the sentence/review
        sent = str(sent, 'utf-8')
        sent = re.sub("[^\w]", " ",  sent).split()

        for word in sent: # for each word in a review/sentence
            try:
                vec = w2vTrained_model.wv[word]
                sent_vec += vec
                cnt_words += 1
            except:
                pass
        sent_vec /= cnt_words
        sent_vectors.append(sent_vec)

    return np.nan_to_num(sent_vectors)
```

In [29]:
```
# W2V Main Function
# W2V Featurisation, Standardisation, Grid Search and Random Search,
# Perturbation test to remove multicollinear features
# Can't find top n words using weight vector.

from sklearn.preprocessing import StandardScaler

# W2V Train
w2v_trainModel = trainW2V_model(X_train['CleanedText'].values)
X_train_vect = computeAvgW2V(w2v_trainModel, X_train['CleanedText'].values)

# W2V Test
w2v_testModel = trainW2V_model(X_test['CleanedText'].values)
X_test_vect = computeAvgW2V(w2v_testModel, X_test['CleanedText'].values)

# Standardisation. Set "with_mean=True" coz W2V vector is dense, not sparse
scaler = StandardScaler(copy=False).fit(X_train_vect)
X_train_vect = scaler.transform(X_train_vect)
scaler = StandardScaler(copy=False).fit(X_test_vect)
X_test_vect = scaler.transform(X_test_vect)

print(bold + "\n\n1) Grid Search and Random Search CV using Logistic Regression"+ end

# Do both grid Search and Random Search.
# The function returns optimal value of lambda
# Last parameter sets the maximum value of C for RandomCV
optimal_lambda, best_regularizer = gridRandomCV(X_train_vect, X_test_vect, 0.01)

# To check sparsity and f1 score for increasing values of lambda
# testL1_increaseLambda(X_train_vect, X_test_vect)

optimal_lambda = 0.00001
# Do pertubation test to check multicollinearity.
# Get weight vector after removing collinear features.
# The last parameter denotes whether train vector is sparse or not
weights_non_collinear, mask = doPertubationTest(
                                X_train_vect, optimal_lambda, best_regularizer, False

print("\nLength of Weight Vector (After Removing Collinearity): "
                                       + str(len(weights_non_collinear)))

# print(w2v_trainModel.vocabulary)


# To print top n words
# find_top_words(count_vect, weights_non_collinear, mask, 20)
```

```
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:20: RuntimeWarning: invalid value encounte
```

1) Grid Search and Random Search CV using Logistic Regression
Best Penalty: l2RandomizedSearchCV: Best C: 0.009682615757193976

```
Mean Test Score (+/-) Standard Deviation for Parameters:
0.860 (+/-0.007) for {'C': 0.00417022004702574, 'penalty': 'l1'}
0.875 (+/-0.013) for {'C': 0.009325573593386588, 'penalty': 'l2'}
0.873 (+/-0.013) for {'C': 0.0030233257263183977, 'penalty': 'l2'}
0.872 (+/-0.014) for {'C': 0.0023608897695197605, 'penalty': 'l2'}
0.873 (+/-0.015) for {'C': 0.001862602113776709, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.0066974603680348, 'penalty': 'l2'}
0.864 (+/-0.008) for {'C': 0.005388167340033569, 'penalty': 'l1'}
0.856 (+/-0.007) for {'C': 0.0031327351693227513, 'penalty': 'l1'}
0.850 (+/-0.003) for {'C': 0.0020445224973151743, 'penalty': 'l1'}
0.851 (+/-0.004) for {'C': 0.0022957721372982554, 'penalty': 'l1'}
0.874 (+/-0.011) for {'C': 0.006704675101784022, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.004572048079869883, 'penalty': 'l2'}
0.849 (+/-0.001) for {'C': 0.0014038693859523377, 'penalty': 'l1'}
0.875 (+/-0.012) for {'C': 0.007783892363365335, 'penalty': 'l2'}
0.875 (+/-0.012) for {'C': 0.009682615757193976, 'penalty': 'l2'}
0.873 (+/-0.015) for {'C': 0.0009280080864073792, 'penalty': 'l2'}
0.868 (+/-0.012) for {'C': 0.008763891522960383, 'penalty': 'l1'}
0.868 (+/-0.011) for {'C': 0.008291469073671415, 'penalty': 'l1'}
0.869 (+/-0.016) for {'C': 0.00039054783232882363, 'penalty': 'l2'}
0.849 (+/-0.000) for {'C': 0.0005924320130156346, 'penalty': 'l1'}
0.873 (+/-0.015) for {'C': 0.000983468338330501, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.006716540974221343, 'penalty': 'l2'}
0.864 (+/-0.008) for {'C': 0.005331652849730171, 'penalty': 'l1'}
0.873 (+/-0.013) for {'C': 0.002896296405458124, 'penalty': 'l2'}
0.866 (+/-0.009) for {'C': 0.006865009276815837, 'penalty': 'l1'}
0.874 (+/-0.012) for {'C': 0.004125388415030261, 'penalty': 'l2'}
0.875 (+/-0.012) for {'C': 0.007501443149449675, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.006606357302380029, 'penalty': 'l2'}
0.873 (+/-0.013) for {'C': 0.002804439920644052, 'penalty': 'l2'}
0.873 (+/-0.014) for {'C': 0.0022212454753537483, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.004478935261759052, 'penalty': 'l2'}
0.849 (+/-0.000) for {'C': 0.0009617226044546168, 'penalty': 'l1'}
0.854 (+/-0.006) for {'C': 0.0028777533858634873, 'penalty': 'l1'}
0.874 (+/-0.012) for {'C': 0.005247993768704455, 'penalty': 'l2'}
0.866 (+/-0.009) for {'C': 0.0067883553293989094, 'penalty': 'l1'}
0.869 (+/-0.011) for {'C': 0.009104483817931015, 'penalty': 'l1'}
0.874 (+/-0.012) for {'C': 0.004915731592803383, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.005659120274641931, 'penalty': 'l2'}
0.850 (+/-0.001) for {'C': 0.0014672857490581016, 'penalty': 'l1'}
0.852 (+/-0.005) for {'C': 0.0026097897961555685, 'penalty': 'l1'}
0.873 (+/-0.015) for {'C': 0.0010233442882782585, 'penalty': 'l2'}
0.875 (+/-0.013) for {'C': 0.009499381385326397, 'penalty': 'l2'}
```

```
0.860 (+/-0.007) for {'C': 0.004141792695269026, 'penalty': 'l1'}
0.867 (+/-0.011) for {'C': 0.007654851044408709, 'penalty': 'l1'}
0.866 (+/-0.009) for {'C': 0.006637946452197888, 'penalty': 'l1'}
0.875 (+/-0.012) for {'C': 0.007924035857960192, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.00586555040501993, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.0054078818960096465, 'penalty': 'l2'}
0.850 (+/-0.001) for {'C': 0.0013927634725075856, 'penalty': 'l1'}
0.874 (+/-0.013) for {'C': 0.0037758434115109465, 'penalty': 'l2'}
0.850 (+/-0.002) for {'C': 0.0016535419711693278, 'penalty': 'l1'}
0.858 (+/-0.007) for {'C': 0.00361261020211621, 'penalty': 'l1'}
0.875 (+/-0.012) for {'C': 0.007508121031361555, 'penalty': 'l2'}
0.849 (+/-0.000) for {'C': 0.0012631488504488832, 'penalty': 'l1'}
0.874 (+/-0.011) for {'C': 0.006236722070556089, 'penalty': 'l2'}
0.857 (+/-0.007) for {'C': 0.0035393908879830195, 'penalty': 'l1'}
0.853 (+/-0.006) for {'C': 0.0026992789176502607, 'penalty': 'l1'}
0.867 (+/-0.011) for {'C': 0.007527883531756042, 'penalty': 'l1'}
0.869 (+/-0.012) for {'C': 0.009648400471483855, 'penalty': 'l1'}
0.863 (+/-0.008) for {'C': 0.004981090696282366, 'penalty': 'l1'}
0.872 (+/-0.015) for {'C': 0.001147459729533752, 'penalty': 'l2'}
0.849 (+/-0.000) for {'C': 0.0006406732983821728, 'penalty': 'l1'}
0.865 (+/-0.010) for {'C': 0.005783896143871318, 'penalty': 'l1'}
0.859 (+/-0.007) for {'C': 0.0037980328676869702, 'penalty': 'l1'}
0.869 (+/-0.011) for {'C': 0.009033795205622539, 'penalty': 'l1'}
0.873 (+/-0.013) for {'C': 0.003013604954098934, 'penalty': 'l2'}
0.865 (+/-0.009) for {'C': 0.006171449136207239, 'penalty': 'l1'}
0.865 (+/-0.009) for {'C': 0.005786339307357229, 'penalty': 'l1'}
0.875 (+/-0.012) for {'C': 0.008859420993107745, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.00470640807662421, 'penalty': 'l2'}
0.865 (+/-0.009) for {'C': 0.006233601157918028, 'penalty': 'l1'}
0.853 (+/-0.006) for {'C': 0.0026900105125782802, 'penalty': 'l1'}
0.874 (+/-0.011) for {'C': 0.006908969175169239, 'penalty': 'l2'}
0.873 (+/-0.014) for {'C': 0.0007008149603262915, 'penalty': 'l2'}
0.850 (+/-0.001) for {'C': 0.0013713574962887776, 'penalty': 'l1'}
0.849 (+/-0.003) for {'C': 0.0019195607823743045, 'penalty': 'l1'}
0.872 (+/-0.014) for {'C': 0.0006600017272206249, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.005082475375672891, 'penalty': 'l2'}
0.875 (+/-0.012) for {'C': 0.009230245355464834, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.0051886187586390195, 'penalty': 'l2'}
0.853 (+/-0.023) for {'C': 0.0001988013383979559, 'penalty': 'l2'}
0.849 (+/-0.000) for {'C': 0.001073653029524273, 'penalty': 'l1'}
0.873 (+/-0.014) for {'C': 0.002462110676030459, 'penalty': 'l2'}
0.873 (+/-0.013) for {'C': 0.002887178327056099, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.005528219786857659, 'penalty': 'l2'}
0.853 (+/-0.006) for {'C': 0.0027831883821020508, 'penalty': 'l1'}
0.873 (+/-0.013) for {'C': 0.002791836790111395, 'penalty': 'l2'}
0.864 (+/-0.009) for {'C': 0.005700666604160945, 'penalty': 'l1'}
0.874 (+/-0.011) for {'C': 0.0056103021925571, 'penalty': 'l2'}
0.875 (+/-0.012) for {'C': 0.008129949703859737, 'penalty': 'l2'}
```

0.872 (+/-0.014) for {'C': 0.002329742738410204, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.00612948092727815, 'penalty': 'l2'}
0.868 (+/-0.011) for {'C': 0.008635418545594287, 'penalty': 'l1'}
0.874 (+/-0.012) for {'C': 0.004278126325923222, 'penalty': 'l2'}
0.850 (+/-0.001) for {'C': 0.0013645522566068501, 'penalty': 'l1'}
0.850 (+/-0.001) for {'C': 0.0015333847791957444, 'penalty': 'l1'}
0.849 (+/-0.000) for {'C': 0.00044551878544761726, 'penalty': 'l1'}
0.875 (+/-0.012) for {'C': 0.00781958462489466, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.007129889803826767, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.006543237716895042, 'penalty': 'l2'}

The optimal value of lambda using RandomizedSearchCV is 103.277877.
Metric Analysis of Logistic Classifier for Optimal Lamdba

Accuracy = 84.250000
Precision = 84.386493
Recall = 99.782566
F1 Score = 91.440993
Confusion Matrix

True Negatives = 7
True Positives = 5048
False Negatives = 11
False Positives = 934

Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 1.0
True Negative Rate(TNR) = 0.01
False Positive Rate(FPR) = 0.99
False Negative Rate(FNR) = 0.0

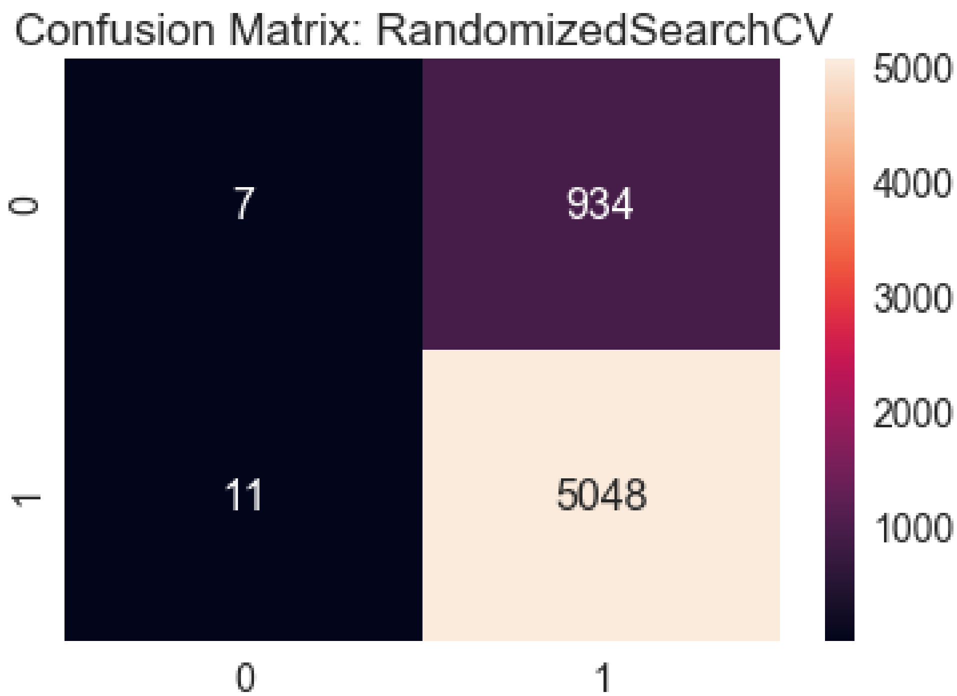
GridSearchCV: Best C: 100

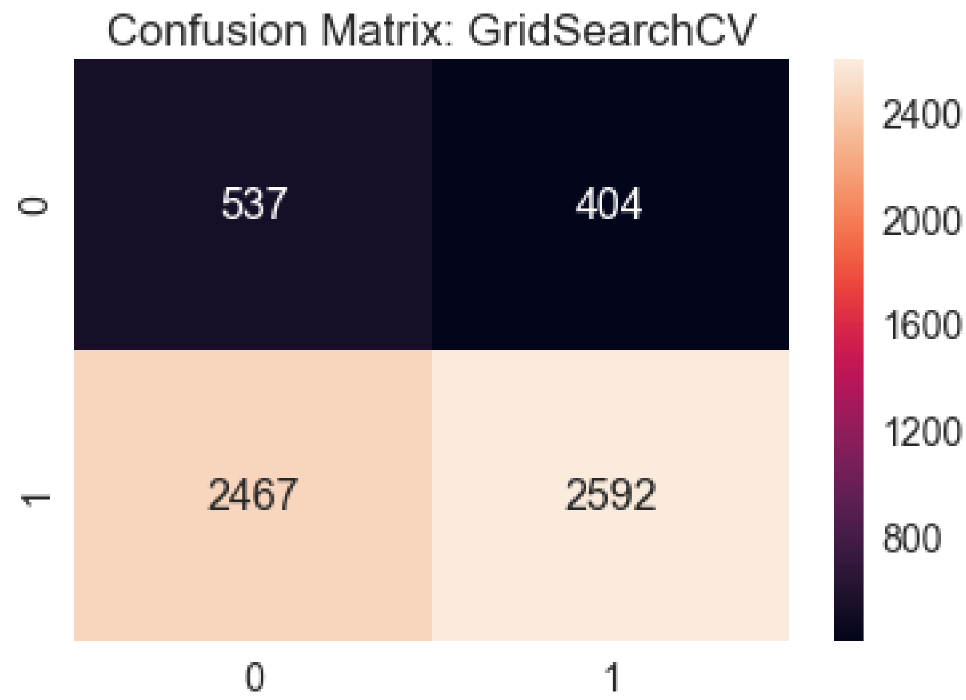
The optimal value of lambda using GridSearchCV is 0.010000.
Metric Analysis of Logistic Classifier for Optimal Lamdba

Accuracy = 52.150000
Precision = 86.515354
Recall = 51.235422
F1 Score = 64.357542
Confusion Matrix

True Negatives = 537
True Positives = 2592
False Negatives = 2467
False Positives = 404

```
Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.51
True Negative Rate(TNR) = 0.57
False Positive Rate(FPR) = 0.43
False Negative Rate(FNR) = 0.49

Length of Weight Vector (Before Removing Collinearity): 300
Distance between Weight vectors before & after Perturbation  = 0.0
Multicollinear Features = 9

Length of Weight Vector (After Removing Collinearity): 291
```





9 TF-ID Weighted W2V

In [30]:
```
# average Word2Vec
# compute average word2vec for each review.

def compute_tfidW2V(w2v_model, model_tf_idf, count_vect, reviewText):

    tfidf_sent_vectors = []; # the tfidf-w2v for each sentence/review is stored in th
    row=0;

    # TF-IDF weighted Word2Vec
    tfidf_feats = count_vect.get_feature_names() # tfidf words/col-names

    # iterate for each review/sentence
    for sent in reviewText:
        sent_vec = np.zeros(w2v_dim) # as word vectors are of zero length
        weight_sum =0; # num of words with a valid vector in the sentence/review
        sent = str(sent, 'utf-8')
        sent = re.sub("[^\w]", " ",  sent).split()

        for word in sent: # for each word in a review/sentence
            try:
                vec = w2v_model.wv[word]
```

```
                # obtain the tf_idfidf of a word in a sentence/review
                tfidf = model_tf_idf[row, tfidf_feats.index(word)]
                sent_vec += (vec * tfidf)
                weight_sum += tfidf
            except:
                pass
        sent_vec /= weight_sum

        tfidf_sent_vectors.append(sent_vec)
        row += 1

    return np.nan_to_num(tfidf_sent_vectors)
```

In [31]:
```
# tf-df weighted W2V Main Function
# tfidf and W2V Featurisation, Standardisation, Grid Search and Random Search,
# Perturbation test to remove multicollinear features
# Can't find top n words using weight vector.

from sklearn.preprocessing import StandardScaler

# TFID
count_vect = TfidfVectorizer(dtype="float") #in scikit-learn
X_train_tfid_vect = count_vect.fit_transform(X_train['CleanedText'].values)

# TFID Test
X_test_tfid_vect = count_vect.transform(X_test['CleanedText'].values)


X_train_vect = compute_tfidW2V(w2v_trainModel, X_train_tfid_vect,
                               count_vect, X_train['CleanedText'].values)
X_test_vect = compute_tfidW2V(w2v_testModel, X_test_tfid_vect,
                              count_vect, X_test['CleanedText'].values)


# Standardisation. Set "with_mean=True" coz W2V vector is dense, not sparse
scaler = StandardScaler(copy=False).fit(X_train_vect)
X_train_vect = scaler.transform(X_train_vect)
scaler = StandardScaler(copy=False).fit(X_test_vect)
X_test_vect = scaler.transform(X_test_vect)

print(bold + "\n\n1) Grid Search and Random Search CV using Logistic Regression"+ end)

# Do both grid Search and Random Search.
# The function returns optimal value of lambda
# sets the maximum value of C to be 10**4 for RandomCV
optimal_lambda, best_regularizer = gridRandomCV(X_train_vect, X_test_vect, 1)

# To check sparsity and f1 score for increasing values of lambda
```

```python
# testL1_increaseLambda(X_train_vect, X_test_vect)

# Do pertubation test to check multicollinearity.
# Get weight vector after removing collinear features.
weights_non_collinear, mask = doPertubationTest(
                                X_train_vect, optimal_lambda, best_regularizer, False)

print("\nLength of Weight Vector (After Removing Collinearity): "
                                    + str(len(weights_non_collinear)))

# To print top n words
# find_top_words(count_vect, weights_non_collinear, mask, 20)
```

```
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:28: RuntimeWarning: invalid value encounte
```

1) Grid Search and Random Search CV using Logistic Regression
Best Penalty: l1RandomizedSearchCV: Best C: 0.9648400471483856

```
Mean Test Score (+/-) Standard Deviation for Parameters:
0.871 (+/-0.010) for {'C': 0.417022004702574, 'penalty': 'l1'}
0.876 (+/-0.013) for {'C': 0.9325573593386588, 'penalty': 'l2'}
0.872 (+/-0.011) for {'C': 0.30233257263183977, 'penalty': 'l2'}
0.871 (+/-0.010) for {'C': 0.23608897695197606, 'penalty': 'l2'}
0.870 (+/-0.010) for {'C': 0.1862602113776709, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.66974603680348, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.538816734003357, 'penalty': 'l1'}
0.870 (+/-0.010) for {'C': 0.3132735169322751, 'penalty': 'l1'}
0.867 (+/-0.008) for {'C': 0.20445224973151743, 'penalty': 'l1'}
0.868 (+/-0.008) for {'C': 0.22957721372982554, 'penalty': 'l1'}
0.876 (+/-0.011) for {'C': 0.6704675101784022, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.45720480798698826, 'penalty': 'l2'}
0.865 (+/-0.010) for {'C': 0.14038693859523377, 'penalty': 'l1'}
0.876 (+/-0.012) for {'C': 0.7783892363365335, 'penalty': 'l2'}
0.876 (+/-0.013) for {'C': 0.9682615757193975, 'penalty': 'l2'}
0.869 (+/-0.009) for {'C': 0.09280080864073792, 'penalty': 'l2'}
0.877 (+/-0.011) for {'C': 0.8763891522960383, 'penalty': 'l1'}
0.877 (+/-0.011) for {'C': 0.8291469073671416, 'penalty': 'l1'}
0.866 (+/-0.009) for {'C': 0.03905478323288236, 'penalty': 'l2'}
0.864 (+/-0.008) for {'C': 0.059243201301563464, 'penalty': 'l1'}
0.869 (+/-0.010) for {'C': 0.0983468338330501, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.6716540974221343, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.5331652849730171, 'penalty': 'l1'}
0.872 (+/-0.011) for {'C': 0.2896296405458124, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.6865009276815837, 'penalty': 'l1'}
0.873 (+/-0.012) for {'C': 0.4125388415030261, 'penalty': 'l2'}
0.876 (+/-0.012) for {'C': 0.7501443149449675, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.6606357302380029, 'penalty': 'l2'}
```

```
0.872 (+/-0.011) for {'C': 0.2804439920644052, 'penalty': 'l2'}
0.871 (+/-0.010) for {'C': 0.2221245475353748, 'penalty': 'l2'}
0.874 (+/-0.012) for {'C': 0.44789352617590517, 'penalty': 'l2'}
0.865 (+/-0.009) for {'C': 0.09617226044546168, 'penalty': 'l1'}
0.869 (+/-0.010) for {'C': 0.28777533858634874, 'penalty': 'l1'}
0.875 (+/-0.011) for {'C': 0.5247993768704455, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.678835532939891, 'penalty': 'l1'}
0.877 (+/-0.011) for {'C': 0.9104483817931015, 'penalty': 'l1'}
0.874 (+/-0.011) for {'C': 0.4915731592803383, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.5659120274641931, 'penalty': 'l2'}
0.866 (+/-0.010) for {'C': 0.14672857490581015, 'penalty': 'l1'}
0.868 (+/-0.009) for {'C': 0.2609789796155568, 'penalty': 'l1'}
0.869 (+/-0.010) for {'C': 0.10233442882782584, 'penalty': 'l2'}
0.876 (+/-0.013) for {'C': 0.9499381385326396, 'penalty': 'l2'}
0.871 (+/-0.010) for {'C': 0.41417926952690265, 'penalty': 'l1'}
0.876 (+/-0.011) for {'C': 0.7654851044408709, 'penalty': 'l1'}
0.876 (+/-0.011) for {'C': 0.6637946452197888, 'penalty': 'l1'}
0.876 (+/-0.012) for {'C': 0.7924035857960191, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.5865550405019929, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.5407881896009646, 'penalty': 'l2'}
0.865 (+/-0.010) for {'C': 0.13927634725075855, 'penalty': 'l1'}
0.873 (+/-0.012) for {'C': 0.3775843411510946, 'penalty': 'l2'}
0.866 (+/-0.010) for {'C': 0.16535419711693278, 'penalty': 'l1'}
0.871 (+/-0.010) for {'C': 0.361261020211621, 'penalty': 'l1'}
0.876 (+/-0.012) for {'C': 0.7508121031361555, 'penalty': 'l2'}
0.865 (+/-0.010) for {'C': 0.1263148850448883, 'penalty': 'l1'}
0.875 (+/-0.011) for {'C': 0.6236722070556089, 'penalty': 'l2'}
0.871 (+/-0.010) for {'C': 0.35393908879830194, 'penalty': 'l1'}
0.868 (+/-0.009) for {'C': 0.2699278917650261, 'penalty': 'l1'}
0.876 (+/-0.011) for {'C': 0.7527883531756042, 'penalty': 'l1'}
0.878 (+/-0.010) for {'C': 0.9648400471483856, 'penalty': 'l1'}
0.873 (+/-0.011) for {'C': 0.49810906962823653, 'penalty': 'l1'}
0.870 (+/-0.010) for {'C': 0.11474597295337519, 'penalty': 'l2'}
0.864 (+/-0.007) for {'C': 0.06406732983821728, 'penalty': 'l1'}
0.875 (+/-0.011) for {'C': 0.5783896143871318, 'penalty': 'l1'}
0.871 (+/-0.010) for {'C': 0.379803286768697, 'penalty': 'l1'}
0.877 (+/-0.011) for {'C': 0.9033795205622538, 'penalty': 'l1'}
0.872 (+/-0.011) for {'C': 0.3013604954098934, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.6171449136207239, 'penalty': 'l1'}
0.875 (+/-0.011) for {'C': 0.5786339307357229, 'penalty': 'l1'}
0.876 (+/-0.013) for {'C': 0.8859420993107745, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.470640807662421, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.6233601157918027, 'penalty': 'l1'}
0.868 (+/-0.009) for {'C': 0.269001051257828, 'penalty': 'l1'}
0.876 (+/-0.011) for {'C': 0.690896917516924, 'penalty': 'l2'}
0.868 (+/-0.009) for {'C': 0.07008149603262914, 'penalty': 'l2'}
0.865 (+/-0.010) for {'C': 0.13713574962887776, 'penalty': 'l1'}
0.867 (+/-0.010) for {'C': 0.19195607823743044, 'penalty': 'l1'}
```

```
0.868 (+/-0.009) for {'C': 0.06600017272206249, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.508247537567289, 'penalty': 'l2'}
0.877 (+/-0.013) for {'C': 0.9230245355464833, 'penalty': 'l2'}
0.875 (+/-0.010) for {'C': 0.518861875863902, 'penalty': 'l2'}
0.865 (+/-0.007) for {'C': 0.01988013383979559, 'penalty': 'l2'}
0.865 (+/-0.009) for {'C': 0.1073653029524273, 'penalty': 'l1'}
0.871 (+/-0.010) for {'C': 0.2462110676030459, 'penalty': 'l2'}
0.872 (+/-0.011) for {'C': 0.2887178327056099, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.5528219786857659, 'penalty': 'l2'}
0.868 (+/-0.010) for {'C': 0.2783188382102051, 'penalty': 'l1'}
0.872 (+/-0.010) for {'C': 0.2791836790111395, 'penalty': 'l2'}
0.874 (+/-0.011) for {'C': 0.5700666604160946, 'penalty': 'l1'}
0.875 (+/-0.011) for {'C': 0.56103021925571, 'penalty': 'l2'}
0.876 (+/-0.012) for {'C': 0.8129949703859737, 'penalty': 'l2'}
0.871 (+/-0.010) for {'C': 0.23297427384102043, 'penalty': 'l2'}
0.875 (+/-0.011) for {'C': 0.6129480927278149, 'penalty': 'l2'}
0.877 (+/-0.011) for {'C': 0.8635418545594287, 'penalty': 'l1'}
0.874 (+/-0.011) for {'C': 0.42781263259232216, 'penalty': 'l2'}
0.865 (+/-0.010) for {'C': 0.13645522566068502, 'penalty': 'l1'}
0.865 (+/-0.010) for {'C': 0.15333847791957445, 'penalty': 'l1'}
0.863 (+/-0.009) for {'C': 0.044551878544761725, 'penalty': 'l1'}
0.876 (+/-0.012) for {'C': 0.7819584624894659, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.7129889803826767, 'penalty': 'l2'}
0.876 (+/-0.011) for {'C': 0.6543237716895042, 'penalty': 'l2'}
```

The optimal value of lambda using RandomizedSearchCV is 1.036441.

Metric Analysis of Logistic Classifier for Optimal Lamdba

```
Accuracy        = 52.566667
Precision        = 83.540467
Recall                = 54.477169
F1 Score        = 65.948792
```

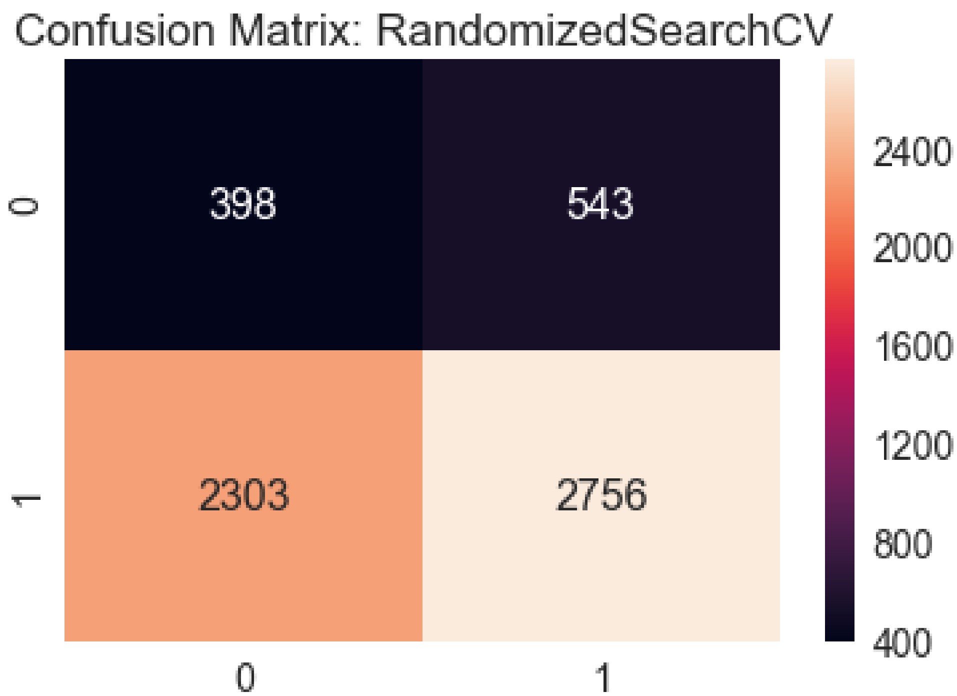
Confusion Matrix

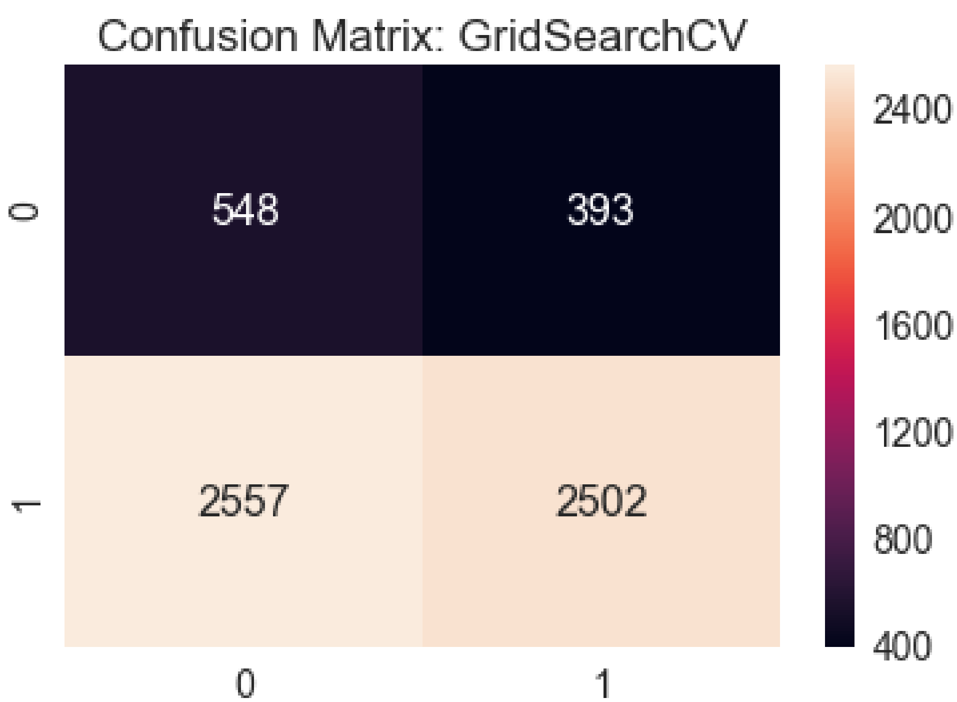
```
True Negatives = 398
True Positives = 2756
False Negatives = 2303
False Positives = 543

Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.54
True Negative Rate(TNR) = 0.42
False Positive Rate(FPR) = 0.58
False Negative Rate(FNR) = 0.46

GridSearchCV: Best C: 100
```

```
The optimal value of lambda using GridSearchCV is 0.010000.
```
Metric Analysis of Logistic Classifier for Optimal Lamdba

```
Accuracy         = 50.833333
Precision         = 86.424870
Recall                 = 49.456414
F1 Score         = 62.911743
```
Confusion Matrix

```
True Negatives = 548
True Positives = 2502
False Negatives = 2557
False Positives = 393

Total Actual Positives = 5059
Total Actual Negatives = 941

True Positive Rate(TPR) = 0.49
True Negative Rate(TNR) = 0.58
False Positive Rate(FPR) = 0.42
False Negative Rate(FNR) = 0.51

Length of Weight Vector (Before Removing Collinearity): 300


C:\Anaconda\lib\site-packages\ipykernel_launcher.py:32: RuntimeWarning: divide by zero encounte
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:32: RuntimeWarning: invalid value encounte
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:42: RuntimeWarning: invalid value encounte
C:\Anaconda\lib\site-packages\ipykernel_launcher.py:42: RuntimeWarning: invalid value encounte


Distance between Weight vectors before & after Perturbation  = 12.02
Multicollinear Features = 136

Length of Weight Vector (After Removing Collinearity): 164
```


Confusion Matrix: RandomizedSearchCV
398
543
2303
2756
0
1
0
1
2400
2000
1600
1200
800
400


Confusion Matrix: GridSearchCV
548
393
2557
2502
0
1
0
1
2400
2000
1600
1200
800
400

10 Summary Statistics

```
In [4]: from IPython.display import Image
        Image(filename='summary.png')
```

Out[4]:

Model	Method	Hyper Parameter	Test Metric
LR on BoW	RandomSearchCV	Lamda = 5.611436	**F1 Score = 94.17.** Accuracy = 90.01
LR on BoW	GridSearchCV	Lamda = 1000	**F1 Score = 94.16.** Accuracy = 89.70
LR on TF-IDF	RandomSearchCV	Lamda = 0.002245	**F1 Score = 91.77.** Accuracy = 86.03
LR on TF-IDF	GridSearchCV	Lamda = 1000	**F1 Score = 93.92.** Accuracy = 89.28
LR on W2V	RandomSearchCV	Lamda = 103.27	**F1 Score = 91.44.** Accuracy = 84.25
LR on W2V	GridSearchCV	Lamda = 0.01	**F1 Score = 64.35.** Accuracy = 52.15
LR on TF-IDF W2V	RandomSearchCV	Lamda = 1.04	**F1 Score = 65.95.** Accuracy = 52.56
LR on BoW	GridSearchCV	Lamda = 0.01	**F1 Score = 62.91.** Accuracy = 50.83

11 Observations

1) From the Sparsity and F1 Score plot, it can be identified that **Performance & Sparsity is the best when Log (Lambda) is between 1 and 2.** i.e. Lambda = 10^1 ~ 10^2 = 10 ~ 100. The lambda values obtained via plotting method is almost same as the lambda value found out by GridSearchCV and RandomSearchCV. (Please note that, **Sparsity = # of non-zero elements**, in this project).

2) It has also been noticed that, **with increasing lambda, the sparsity (# of non-zero elements) has been decreasing steadily.** This is an expected behaviour, as **L1 regularization** is used.

3) The Lambda values found by GridSearchCV and RandomizedSearchCV are near, only when the range of "C" values is set within a narrow range, around optimum. i.e. if the optimal C = 1 (as per GridSearchCV), then by setting C as a uniform distribution between 0 and 4 will yield C = 1 (+/- 0.05) approximately, within say, 100 iterations. But if C value is set as a uniform distribution between 0 and say, 10000, then the error in C value is found to be very high.

4) Alternatively, **if the range of C value is wide, to arrive at optimal C, we need to increase the number of iterations** significantly. It is seen that, when iterations are increased from 100 to 1000, the C value is converging to optimum. But the **time complexity of such an approach would be much higher.**

5) Because of 3 and 4, it is suggested to **use GridSearchCV for faster convergence when the number of dimensions are less.** But, when the # of hyperparameters increase, the # of times the model needs to be trained, increases exponentially. If there are k hyperparameters, then m^k trainings would be required. Hence, **grid search is not good when hyperparameters are more.** In Logistic Regression, there could be only 2 hyperparameters. But there are cases in deep learning where there are 10s or 100s of hyper parameters.

6) **Random Search** is almost as good as Grid search, and also **faster than Grid search when # of hyper parameters is large.** But since the number of iterations required to find the optimal lambda for multiple dimensions is much more, more processing power may be required. Still, it would perform better than the exponential time requirement of Grid Search.

7) The elements of **W2V vector doesnt correspond to each word feature, like in the BoW vector or TF-ID vector.** Hence the weight vector w, that you get, which would be of the same length as W2V vector, once you fit logistic regression, doesnt correlate to word features. **Hence, we cannot find the top 'n' words when we use Word2Vec based featurization.** But we can still find the top 'n' features based on the weight vector, but that do not correspond to any word, hence not interpretable.

8) The best method is found to be **Logistic Regression on Bag of Words.** This method has the highest F1 Score, amongst all the 4 methods. Hence, Bag of Words featurization with Logistic Regression is the classifer of choice.