In [1]:
```python
import warnings
warnings.filterwarnings('ignore')

# From Example Code  https://github.com/Swathiu/Detecting-Fake-Reviews/blob/master/Deception_Detection.py
import pandas as pd
import numpy as np

from nltk.corpus import stopwords
from nltk.tokenize import RegexpTokenizer

from datetime import datetime
from time import time

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, pairwise_distances
from sklearn.metrics import confusion_matrix

import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm

import spacy
```

In [2]:
```python
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
pd.set_option('display.max_colwidth', None)
```

In [3]:
```python
file_path = "C:/Users/tsaie/OneDrive/Desktop/000 Resumes & Projects/# Projects/DS3 Fake Amazon Reviews/Dataset/"
apparel = pd.read_csv(file_path + 'amazon_reviews_us_Apparel_v1_00.tsv.gz', compression='gzip', header=0, sep='\t', quotechar='"', error_bad_lines=False, warn_bad_lines=False, nrows=10_000)
# electronics = pd.read_csv(file_path + 'amazon_reviews_us_Apparel_v1_00.tsv.gz', compression='gzip', header=0, sep='\t', quotechar='"', error_bad_lines=False, warn_bad_lines=False)
# apparel
```

In [4]:
```python
apparel_small = apparel[['verified_purchase', 'review_body']]
# apparel_small
```

Data Cleaning

In [5]:
```python
def data_cleaning(df):
    print("######## Cleaning Data ########")

    # Removing emtpy cells
    df.dropna(inplace=True)

    # Pre-processing Text Reviews

    # Lowercase Words
    df['review_body'] = df['review_body'].apply(
        lambda x: x.lower())

    print("\n######## Lowercase Complete ########")

    # Remove Stop Words. Also remove "br " (HTML line break symbols) and "&#34" (HTML quote symbols)
    stop = stopwords.words('english')
    stop += ["br", "&#34"]

    df['review_body'] = df['review_body'].apply(
        lambda x: ' '.join([word for word in x.split() if word.strip() not in stop]))

    df['review_body'] = df['review_body'].apply(
        lambda x: x.replace("<br /><br />", " "))

    print("\n######## Remove Stop Words Complete ########")

    # Remove Punctuations
    tokenizer = RegexpTokenizer(r'\w+')
    df['review_body'] = df['review_body'].apply(
        lambda x: ' '.join([word for word in tokenizer.tokenize(x)]))

    print("\n######## Remove Punctuation Complete ########")

    # Lemmatization using .lemma_
    nlp = spacy.load('en_core_web_sm', disable=['parser', 'ner'])
    df['review_body'] = df['review_body'].apply(
        lambda x: ' '.join([token.lemma_ for token in nlp(x)]))

    print("\n######## Data Cleaning Complete ########")

    return df
```

In [6]:
```python
apparel_cleaned = data_cleaning(apparel_small)
# apparel_cleaned
```

```
######## Cleaning Data ########

######## Lowercase Complete ########

######## Remove Stop Words Complete ########

######## Remove Punctuation Complete ########

######## Data Cleaning Complete ########
```

Add Bigrams

In [7]:
```python
# https://stackoverflow.com/questions/48331315/how-to-extract-all-the-ngrams-from-a-text-dataframe-column-in-different-order-in

from collections import Counter
from nltk import ngrams
from itertools import chain

def find_ngrams(input_list, n):
    return list(zip(*[input_list[i:] for i in range(n)]))


apparel_cleaned['bigrams'] = apparel_cleaned['review_body'].map(lambda x: find_ngrams(x.split(" "), 2))
# apparel_cleaned
```

In [8]:
```python
apparel_un_verified = apparel_cleaned[apparel_cleaned['verified_purchase'] == 'N']
apparel_verified = apparel_cleaned[apparel_cleaned['verified_purchase'] == 'Y']

# apparel_un_verified
```

In [9]:
```python
verified_bigrams = apparel_verified['bigrams'].tolist()
verified_bigrams = list(chain(*verified_bigrams))

verified_bigram_counts = Counter(verified_bigrams)
verified_bigram_counts.most_common(20)
```

Out[9]:
```
[(('I', 'm'), 928),
 (('love', 'it'), 419),
 (('fit', 'perfectly'), 238),
 (('fit', 'great'), 224),
 (('look', 'like'), 217),
 (('m', '5'), 215),
 (('fit', 'well'), 209),
 (('good', 'quality'), 182),
 (('fit', 'perfect'), 172),
 (('can', 't'), 168),
 (('well', 'make'), 164),
 (('run', 'small'), 163),
 (('order', 'size'), 156),
 (('like', 'picture'), 148),
 (('look', 'great'), 142),
 (('I', 've'), 138),
 (('love', 'shirt'), 127),
 (('fit', 'like'), 125),
 (('order', 'large'), 123),
 (('love', 'dress'), 120)]
```

In [10]:
```python
un_verified_bigrams = apparel_un_verified['bigrams'].tolist()
un_verified_bigrams = list(chain(*un_verified_bigrams))

un_verified_bigram_counts = Counter(un_verified_bigrams)
un_verified_bigram_counts.most_common(20)
```

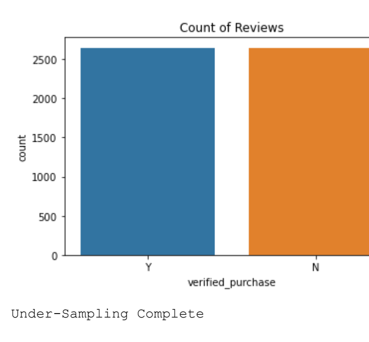
Out[10]:
```
[(('exchange', 'honest'), 446),
 (('I', 'm'), 442),
 (('receive', 'product'), 359),
 (('honest', 'review'), 340),
 (('unbiased', 'review'), 287),
 (('I', 'receive'), 282),
 (('honest', 'unbiased'), 264),
 (('product', 'discount'), 262),
 (('br', 'I'), 240),
 (('discount', 'exchange'), 219),
 (('love', 'it'), 180),
 (('well', 'make'), 173),
 (('discount', 'price'), 153),
 (('fit', 'perfectly'), 148),
 (('I', 've'), 135),
 (('fit', 'well'), 130),
 (('t', 'shirt'), 119),
 (('price', 'exchange'), 112),
 (('good', 'quality'), 112),
 (('look', 'great'), 101)]
```

Feature Engineering + Prepare Data for Machine Learning

In [11]:
```python
def under_sampling(df):
    print("Under-Sampling Data")
    # Count of Reviews
    print("Verified:", len(df[(df['verified_purchase'] == 'Y')]))
    print("Un-Verified:", len(df[(df['verified_purchase'] == 'N')]))

    sample_size = len(df[(df['verified_purchase'] == 'N')])

    authentic_reviews_df = df[df['verified_purchase'] == 'Y']
    fake_reviews_df = df[df['verified_purchase'] == 'N']

    authentic_reviews_us_df = authentic_reviews_df.sample(sample_size)
    under_sampled_df = pd.concat([authentic_reviews_us_df, fake_reviews_df], axis=0)

    print("Under-Sampled Verified", len(under_sampled_df[(under_sampled_df['verified_purchase'] == 'Y')]))
    print("Under-Sampled Un-Verified", len(under_sampled_df[(under_sampled_df['verified_purchase'] == 'N')]))


    # Graph of Data Distribution
    fig, ax = plt.subplots(figsize=(6, 4))
    sns.countplot(x='verified_purchase', data=under_sampled_df)
    plt.title("Count of Reviews")
    plt.show()
    print("Under-Sampling Complete")
    return under_sampled_df
```

In [12]:
```python
apparel_equal_weight = under_sampling(apparel_cleaned)
# apparel_equal_weight
```

```
Under-Sampling Data
Verified: 7353
Un-Verified: 2646
Under-Sampled Verified 2646
Under-Sampled Un-Verified 2646
```




```
Under-Sampling Complete
```

Let's call the bigrams in vertified reviews "gold_bigrams" and the bigrams in unverified reviews "fake_bigrams"

count = number of gold/fake_bigrams in a review

percent = number of gold/fake_bigrams as a percentage of total number of bigrams in a review.

simple score = sum of the gold/fake_bigrams' popularity scores (calculated using simply the bigram's count in the Counter)

normalized score = simple score / total bigram count

In [13]:
```python
def get_bigram_count(bigrams, bigram_dict):
    count = 0
    for bigram in bigrams:
        if bigram in bigram_dict.keys():
            count += 1
    return count

def get_bigram_simple_score(bigrams, bigram_dict):
    score = 0
    for bigram in bigrams:
        if bigram in bigram_dict.keys():
            score += bigram_dict[bigram]
    return score
```

In [14]:
```python
apparel_equal_weight['bigram_count'] = apparel_equal_weight['bigrams'].apply(
    lambda x: len(x))
```

In [15]:
```python
# fake

fake_bigram_dict = dict(un_verified_bigram_counts) # fake_bigram_dict = dict(un_verified_bigram_counts.most_common(30))

apparel_equal_weight['fake_bigram_count'] = apparel_equal_weight['bigrams'].apply(
    lambda x: get_bigram_count(x, fake_bigram_dict))

apparel_equal_weight['fake_bigram_percent'] = apparel_equal_weight['fake_bigram_count'] / apparel_equal_weight['bigram_count']

apparel_equal_weight['fake_bigram_simple_score'] = apparel_equal_weight['bigrams'].apply(
    lambda x: get_bigram_simple_score(x, fake_bigram_dict))

apparel_equal_weight['fake_bigram_normalized_score'] = apparel_equal_weight['fake_bigram_simple_score'] / apparel_equal_weight['bigram_count']
```

In [16]:
```python
# gold

gold_bigram_dict = dict(verified_bigram_counts) # gold_bigram_dict = dict(verified_bigram_counts.most_common(30))

apparel_equal_weight['gold_bigram_count'] = apparel_equal_weight['bigrams'].apply(
    lambda x: get_bigram_count(x, gold_bigram_dict))

apparel_equal_weight['gold_bigram_percent'] = apparel_equal_weight['gold_bigram_count'] / apparel_equal_weight['bigram_count']

apparel_equal_weight['gold_bigram_simple_score'] = apparel_equal_weight['bigrams'].apply(
    lambda x: get_bigram_simple_score(x, gold_bigram_dict))

apparel_equal_weight['gold_bigram_normalized_score'] = apparel_equal_weight['gold_bigram_simple_score'] / apparel_equal_weight['bigram_count']
```

In [22]:
```python
apparel_equal_weight = apparel_equal_weight.fillna(0)
apparel_equal_weight.head(1)
```

Out[22]:

	verified_purchase	review_body	bigrams	bigram_count	fake_bigram_count	fake_bigram_percent	fake_bigram_simple_score	fake_
2976	Y	would ve fit accord size guide would ve perfect drop up cuz wayyyyyy nbig put tight measure order still way big reorder time probably go two size small	[(would, ve), (ve, fit), (fit, accord), (accord, size), (size, guide), (guide, would), (would, ve), (ve, perfect), (perfect, drop), (drop, up), (up, cuz), (cuz, wayyyyyy), (wayyyyyy, nbig), (nbig, put), (put, tight), (tight, measure), (measure, order), (order, still), (still, way), (way, big), (big, reorder), (reorder, time), (time, probably), (probably, go), (go, two), (two, size), (size, small)]	27	10	0.37037	66	

In [23]:
```python
apparel_equal_weight[apparel_equal_weight['verified_purchase'] == 'N'].head(1)
```

Out[23]:

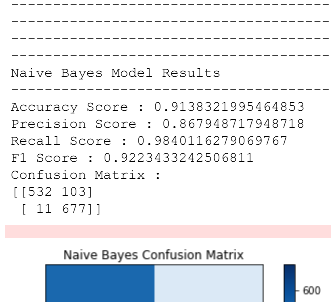
	verified_purchase	review_body	bigrams	bigram_count	fake_bigram_count	fake_bigram_percent	fake_bigram_simple_score	fake_
28	N	second tank order wear often wash well receive many compliment they	[(second, tank), (tank, order), (order, wear), (wear, often), (often, wash), (wash, well), (well, receive), (receive, many), (many, compliment), (compliment, they)]	10	10	1.0	74	

Use Machine Learning to Make Predictions for Verified VS. Unverified

In [19]:
```python
def semi_supervised_learning(df, model, algorithm, threshold=0.8, iterations=40):
    df = df.copy()

    df_unlabled = df[['fake_bigram_count', 'fake_bigram_percent', 'fake_bigram_simple_score', 'fake_bigram_normalized_score',
              'gold_bigram_count', 'gold_bigram_percent', 'gold_bigram_simple_score', 'gold_bigram_normalized_score']]

    df['verified_purchase'] = df['verified_purchase'].apply(lambda x: 1 if x == 'Y' else 0)
    print("Training " + algorithm + " Model")
    labels = df['verified_purchase']

    train_data, test_data, train_label, test_label = train_test_split(df_unlabled, labels, test_size=0.25, random_state=42)

    test_data_copy = test_data.copy()
    test_label_copy = test_label.copy()

    all_labeled = False

    current_iteration = 0

    pbar = tqdm(total=iterations)

    while not all_labeled and (current_iteration < iterations):
        current_iteration += 1
        model.fit(train_data, train_label)

        probabilities = model.predict_proba(test_data)
        pseudo_labels = model.predict(test_data)

        indices = np.argwhere(probabilities > threshold)

        for item in indices:
            train_data.loc[test_data.index[item[0]]] = test_data.iloc[item[0]]
            train_label.loc[test_data.index[item[0]]] = pseudo_labels[item[0]]
        test_data.drop(test_data.index[indices[:, 0]], inplace=True)
        test_label.drop(test_label.index[indices[:, 0]], inplace=True)

        print("--" * 20)

        if len(test_data) == 0:
            print("Exiting loop")
            all_labeled = True
        pbar.update(1)

    pbar.close()
    predicted_labels = model.predict(test_data_copy)

    print(algorithm + ' Model Results')
    print('--' * 20)
    print('Accuracy Score : ' + str(accuracy_score(test_label_copy, predicted_labels)))
    print('Precision Score : ' + str(precision_score(test_label_copy, predicted_labels, pos_label=1)))
    print('Recall Score : ' + str(recall_score(test_label_copy, predicted_labels, pos_label=1)))
    print('F1 Score : ' + str(f1_score(test_label_copy, predicted_labels, pos_label=1)))
    print('Confusion Matrix : \n' + str(confusion_matrix(test_label_copy, predicted_labels)))
    plot_confusion_matrix(test_label_copy, predicted_labels, classes=[1, 0],
                          title=algorithm + ' Confusion Matrix').show()


def plot_confusion_matrix(y_true, y_pred, classes, title=None, cmap=plt.cm.Blues):
    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           xticklabels=classes,
           yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()

    return plt
```

In [20]:
```python
start_time = time()
rf = RandomForestClassifier(random_state=42, criterion='entropy', max_depth=14, max_features='auto', n_estimators=500)
semi_supervised_learning(apparel_equal_weight, model=rf, threshold=0.7, iterations=15, algorithm='Random Forest')
end_time = time()

print("Time taken : ", end_time - start_time)
```

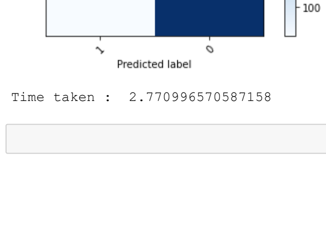
```
  0%|          | 0/15 [00:00<?, ?it/s]
Training Random Forest Model
  7%|█         | 1/15 [00:03<00:52,  3.73s/it]
----------------------------------------
 13%|█         | 2/15 [00:05<00:40,  3.09s/it]
----------------------------------------
 20%|██        | 3/15 [00:06<00:31,  2.63s/it]
----------------------------------------
 27%|███       | 4/15 [00:08<00:25,  2.32s/it]
----------------------------------------
 33%|███       | 5/15 [00:10<00:20,  2.09s/it]
----------------------------------------
 40%|████      | 6/15 [00:11<00:17,  1.95s/it]
----------------------------------------
 47%|█████     | 7/15 [00:13<00:14,  1.84s/it]
----------------------------------------
 53%|█████     | 8/15 [00:14<00:12,  1.76s/it]
----------------------------------------
 60%|██████    | 9/15 [00:16<00:10,  1.71s/it]
----------------------------------------
 67%|███████   | 10/15 [00:17<00:08,  1.67s/it]
----------------------------------------
 73%|███████   | 11/15 [00:19<00:06,  1.64s/it]
----------------------------------------
 80%|████████  | 12/15 [00:21<00:04,  1.62s/it]
----------------------------------------
 87%|█████████ | 13/15 [00:22<00:03,  1.60s/it]
----------------------------------------
 93%|█████████ | 14/15 [00:24<00:01,  1.59s/it]
----------------------------------------
100%|██████████| 15/15 [00:25<00:00,  1.72s/it]
----------------------------------------
Random Forest Model Results
----------------------------------------
Accuracy Score : 0.9606953892668179
Precision Score : 0.9404432132963989
Recall Score : 0.9869186046511628
F1 Score : 0.9631205673758866
Confusion Matrix :
[[592  43]
 [  9 679]]
```




```
Time taken :  26.026597499847412
```

In [21]:
```python
start_time = time()
nb = GaussianNB()
semi_supervised_learning(apparel_equal_weight, model=nb, threshold=0.7, iterations=15, algorithm='Naive Bayes')
end_time = time()

print("Time taken : ", end_time - start_time)
```

```
  0%|          | 0/15 [00:00<?, ?it/s]
Training Naive Bayes Model
100%|██████████| 15/15 [00:02<00:00,  5.66it/s]
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
----------------------------------------
Naive Bayes Model Results
----------------------------------------
Accuracy Score : 0.9138321995464853
Precision Score : 0.867948717948718
Recall Score : 0.9840116279069767
F1 Score : 0.9223433242506811
Confusion Matrix :
[[532 103]
 [ 11 677]]
```




```
Time taken :  2.770996570587158
```

In [ ]: