C1_W2_lecture_nb_01_visualizing_naive_bayes

January 23, 2022

1 Visualizing Naive Bayes

In this lab, we will cover an essential part of data analysis that has not been included in the lecture videos. As we stated in the previous module, data visualization gives insight into the expected performance of any model.

In the following exercise, you are going to make a visual inspection of the tweets dataset using the Naïve Bayes features. We will see how we can understand the log-likelihood ratio explained in the videos as a pair of numerical features that can be fed in a machine learning algorithm.

At the end of this lab, we will introduce the concept of **confidence ellipse** as a tool for representing the Naïve Bayes model visually.

[1]:
```
import numpy as np # Library for linear algebra and math utils
import pandas as pd # Dataframe library

import matplotlib.pyplot as plt # Library for plots
from utils import confidence_ellipse # Function to add confidence ellipses to␣
↪charts
```

Calculate the likelihoods for each tweet

For each tweet, we have calculated the likelihood of the tweet to be positive and the likelihood to be negative. We have calculated in different columns the numerator and denominator of the likelihood ratio introduced previously.

$$log\frac{P(tweet|pos)}{P(tweet|neg)} = log(P(tweet|pos)) - log(P(tweet|neg))$$

$$positive = log(P(tweet|pos)) = \sum_{i=0}^{n} logP(W_i|pos)$$

$$negative = log(P(tweet|neg)) = \sum_{i=0}^{n} logP(W_i|neg)$$

We did not include the code because this is part of this week's assignment. The **'bayes_features.csv'** file contains the final result of this process.

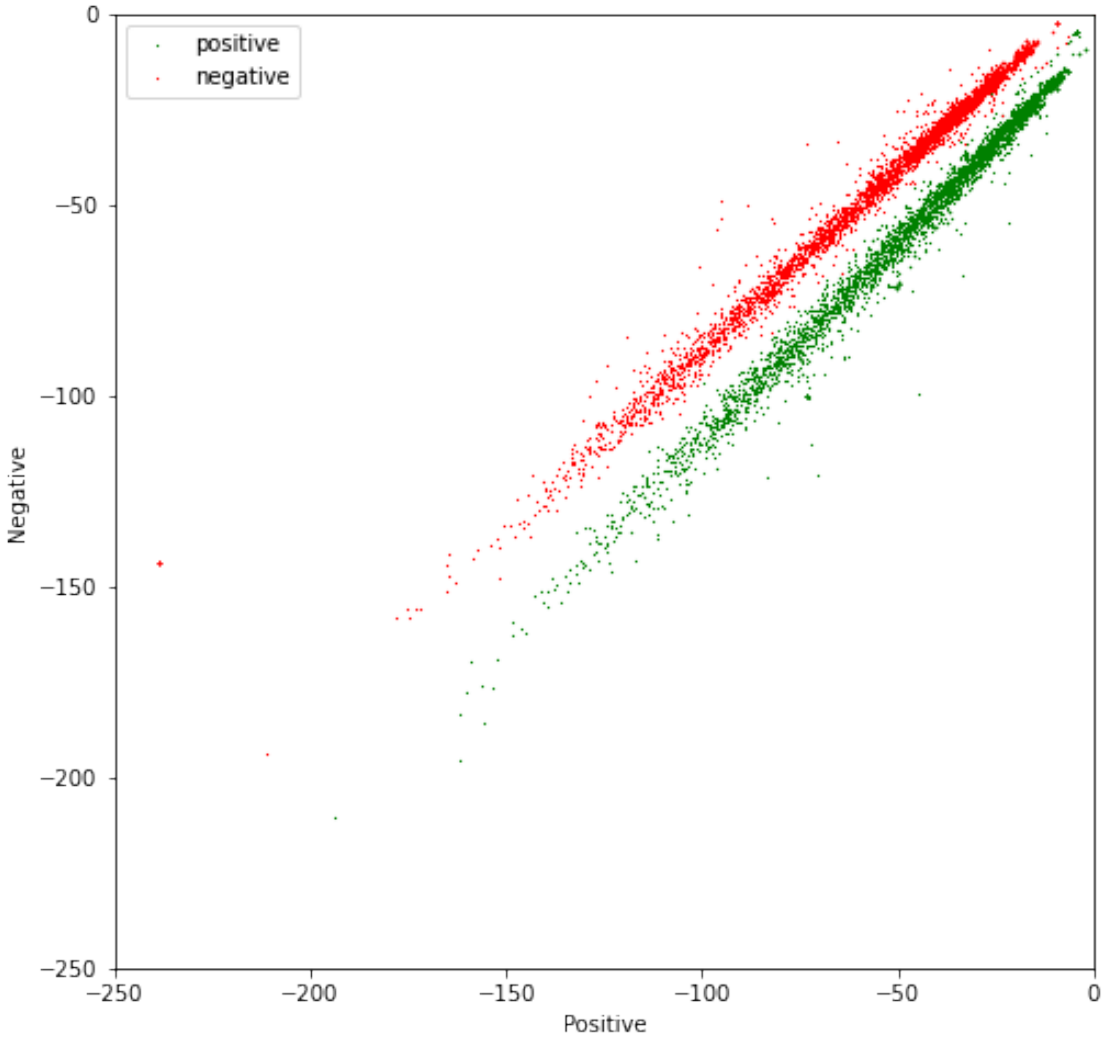
The cell below loads the table in a dataframe. Dataframes are data structures that simplify the manipulation of data, allowing filtering, slicing, joining, and summarization.

[2]:
```
data = pd.read_csv('./data/bayes_features.csv'); # Load the data from the csv␣
↪file

data.head(5) # Print the first 5 tweets features. Each row represents a tweet
```

[2]:

	positive	negative	sentiment
0	-45.763393	-63.351354	1.0
1	-105.491568	-114.204862	1.0
2	-57.028078	-67.216467	1.0
3	-10.055885	-18.589057	1.0
4	-125.749270	-138.334845	1.0

[3]:
```
# Plot the samples using columns 1 and 2 of the matrix
fig, ax = plt.subplots(figsize = (8, 8)) #Create a new figure with a custom size

colors = ['red', 'green'] # Define a color palete
sentiments = ['negative', 'positive']

index = data.index

# Color base on sentiment
for sentiment in data.sentiment.unique():
    ix = index[data.sentiment == sentiment]
    ax.scatter(data.iloc[ix].positive, data.iloc[ix].negative,␣
↪c=colors[int(sentiment)], s=0.1, marker='*',␣
↪label=sentiments[int(sentiment)])

ax.legend(loc='best')

# Custom limits for this chart
plt.xlim(-250,0)
plt.ylim(-250,0)

plt.xlabel("Positive") # x-axis label
plt.ylabel("Negative") # y-axis label
plt.show()
```



2 Using Confidence Ellipses to interpret Naïve Bayes

In this section, we will use the confidence ellipse to give us an idea of what the Naïve Bayes model see.

A confidence ellipse is a way to visualize a 2D random variable. It is a better way than plotting the points over a cartesian plane because, with big datasets, the points can overlap badly and hide the real distribution of the data. Confidence ellipses summarize the information of the dataset with only four parameters:

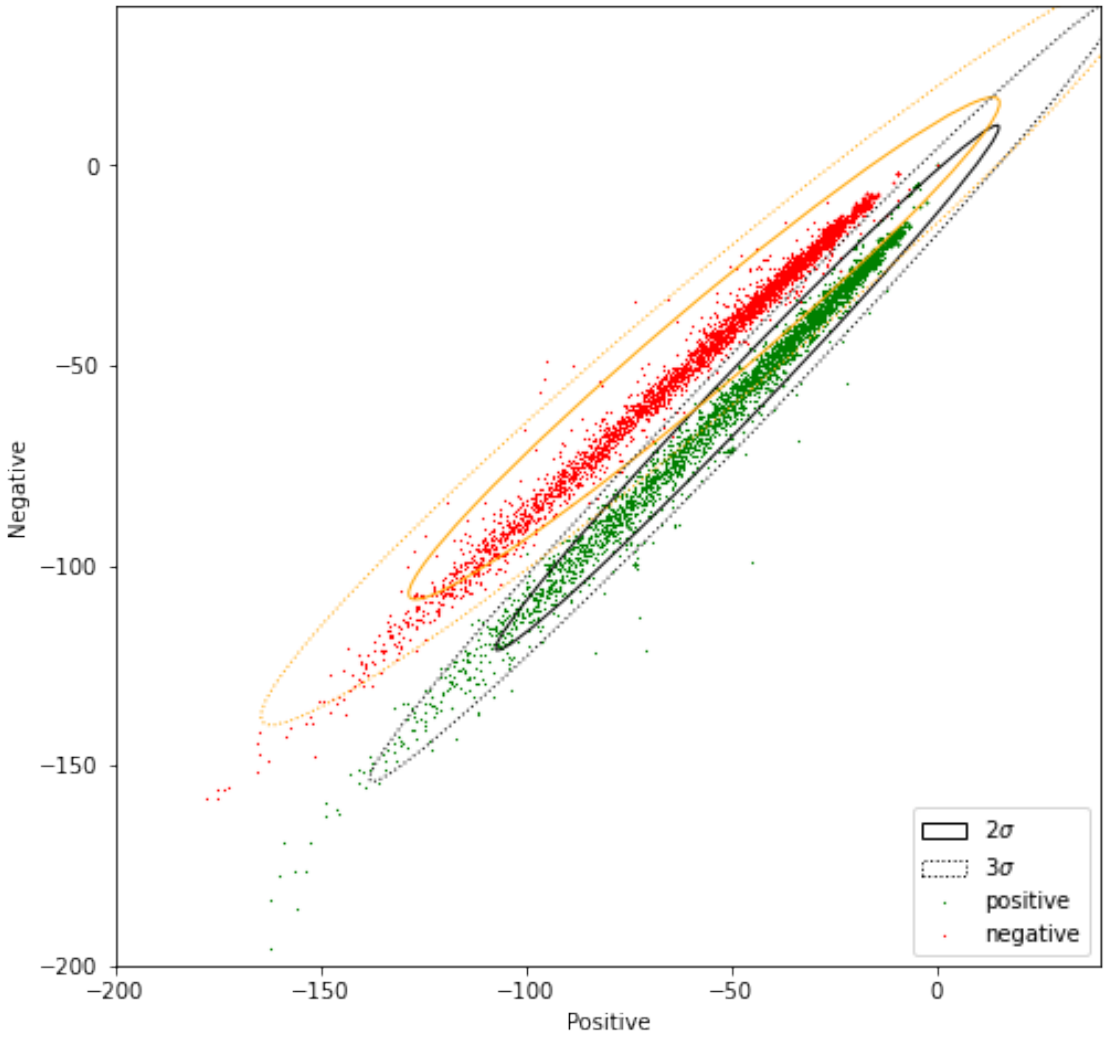
- Center: It is the numerical mean of the attributes
- Height and width: Related with the variance of each attribute. The user must specify the desired amount of standard deviations used to plot the ellipse.
- Angle: Related with the covariance among attributes.

The parameter **n_std** stands for the number of standard deviations bounded by the ellipse. Remember that for normal random distributions:

- About 68% of the area under the curve falls within 1 standard deviation around the mean.
- About 95% of the area under the curve falls within 2 standard deviations around the mean.
- About 99.7% of the area under the curve falls within 3 standard deviations around the mean.

In the next chart, we will plot the data and its corresponding confidence ellipses using 2 std and 3 std.

[4]:
```
# Plot the samples using columns 1 and 2 of the matrix
fig, ax = plt.subplots(figsize = (8, 8))

colors = ['red', 'green'] # Define a color palete
sentiments = ['negative', 'positive']
index = data.index

# Color base on sentiment
for sentiment in data.sentiment.unique():
    ix = index[data.sentiment == sentiment]
    ax.scatter(data.iloc[ix].positive, data.iloc[ix].negative,␣
 ↪c=colors[int(sentiment)], s=0.1, marker='*',␣
 ↪label=sentiments[int(sentiment)])

# Custom limits for this chart
plt.xlim(-200,40)
plt.ylim(-200,40)

plt.xlabel("Positive") # x-axis label
plt.ylabel("Negative") # y-axis label

data_pos = data[data.sentiment == 1] # Filter only the positive samples
data_neg = data[data.sentiment == 0] # Filter only the negative samples

# Print confidence ellipses of 2 std
confidence_ellipse(data_pos.positive, data_pos.negative, ax, n_std=2,␣
 ↪edgecolor='black', label=r'$2\sigma$' )
confidence_ellipse(data_neg.positive, data_neg.negative, ax, n_std=2,␣
 ↪edgecolor='orange')

# Print confidence ellipses of 3 std
confidence_ellipse(data_pos.positive, data_pos.negative, ax, n_std=3,␣
 ↪edgecolor='black', linestyle=':', label=r'$3\sigma$')
confidence_ellipse(data_neg.positive, data_neg.negative, ax, n_std=3,␣
 ↪edgecolor='orange', linestyle=':')
ax.legend(loc='lower right')

plt.show()
```

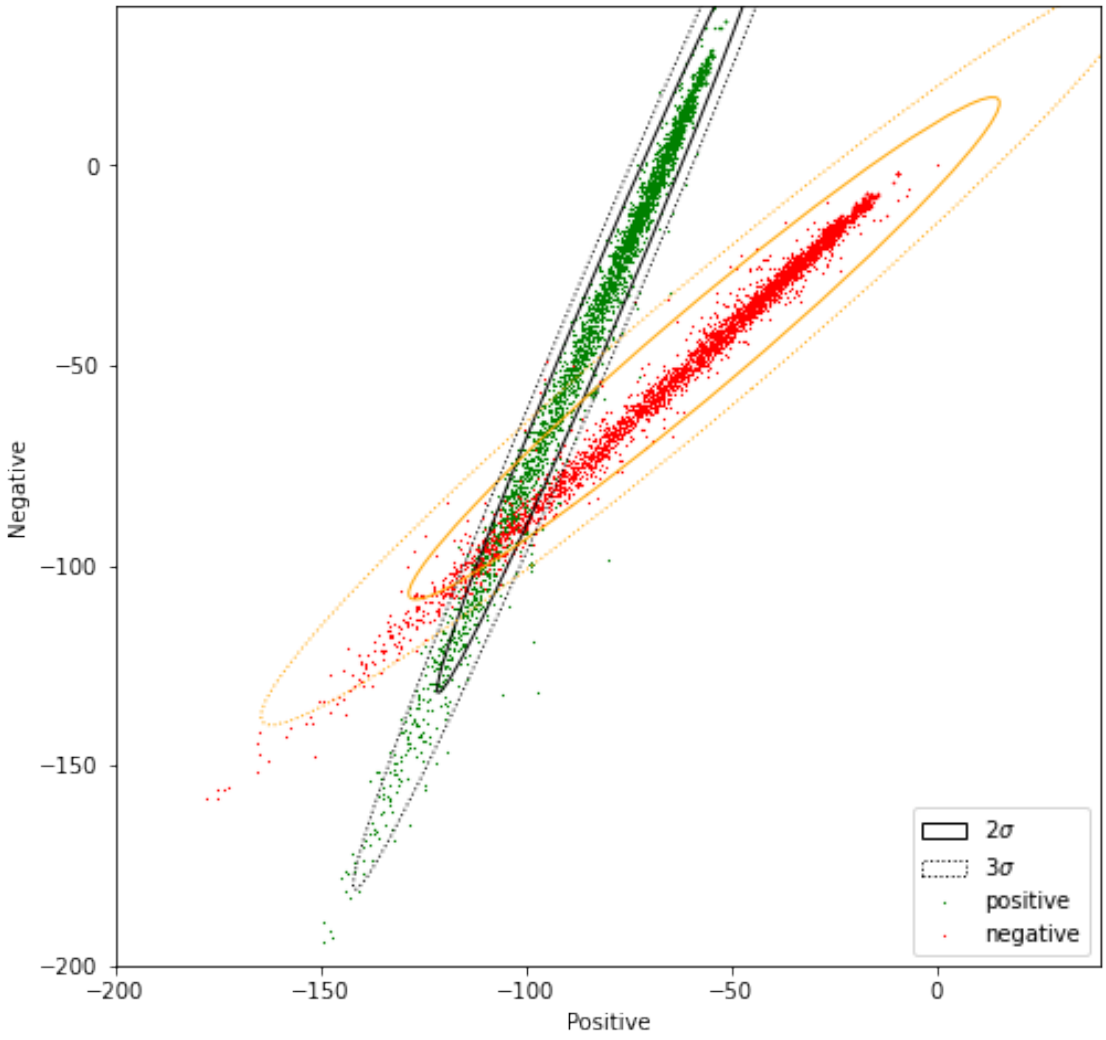


In the next cell, we will modify the features of the samples with positive sentiment (1), in a way that the two distributions overlap. In this case, the Naïve Bayes method will produce a lower accuracy than with the original data.

[5]:
```
data2 = data.copy() # Copy the whole data frame

# The following 2 lines only modify the entries in the data frame where␣
 ↪sentiment == 1
data2.negative[data.sentiment == 1] =  data2.negative * 1.5 + 50 # Modify the␣
 ↪negative attribute
data2.positive[data.sentiment == 1] =  data2.positive / 1.5 - 50 # Modify the␣
 ↪positive attribute
```

Now let us plot the two distributions and the confidence ellipses

[6]:
```
# Plot the samples using columns 1 and 2 of the matrix
fig, ax = plt.subplots(figsize = (8, 8))

colors = ['red', 'green'] # Define a color palete
sentiments = ['negative', 'positive']
index = data2.index

# Color base on sentiment
for sentiment in data2.sentiment.unique():
    ix = index[data2.sentiment == sentiment]
    ax.scatter(data2.iloc[ix].positive, data2.iloc[ix].negative,␣
 ↪c=colors[int(sentiment)], s=0.1, marker='*',␣
 ↪label=sentiments[int(sentiment)])

#ax.scatter(data2.positive, data2.negative, c=[colors[int(k)] for k in data2.
 ↪sentiment], s = 0.1, marker='*')  # Plot a dot for tweet
# Custom limits for this chart
plt.xlim(-200,40)
plt.ylim(-200,40)

plt.xlabel("Positive") # x-axis label
plt.ylabel("Negative") # y-axis label

data_pos = data2[data2.sentiment == 1] # Filter only the positive samples
data_neg = data[data2.sentiment == 0] # Filter only the negative samples

# Print confidence ellipses of 2 std
confidence_ellipse(data_pos.positive, data_pos.negative, ax, n_std=2,␣
 ↪edgecolor='black', label=r'$2\sigma$' )
confidence_ellipse(data_neg.positive, data_neg.negative, ax, n_std=2,␣
 ↪edgecolor='orange')

# Print confidence ellipses of 3 std
confidence_ellipse(data_pos.positive, data_pos.negative, ax, n_std=3,␣
 ↪edgecolor='black', linestyle=':', label=r'$3\sigma$')
confidence_ellipse(data_neg.positive, data_neg.negative, ax, n_std=3,␣
 ↪edgecolor='orange', linestyle=':')
ax.legend(loc='lower right')

plt.show()
```



To give away: Understanding the data allows us to predict if the method will perform well or not. Alternatively, it will allow us to understand why it worked well or bad.

utils

January 23, 2022

[ ]:
```
import re
import string
```

[ ]:
```
import numpy as np
```

[ ]:
```
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.tokenize import TweetTokenizer
```

[ ]:
```
from matplotlib.patches import Ellipse
import matplotlib.transforms as transforms
```

[ ]:
```
def process_tweet(tweet):
    '''
    Input:
        tweet: a string containing a tweet
    Output:
        tweets_clean: a list of words containing the processed tweet

    '''
    stemmer = PorterStemmer()
    stopwords_english = stopwords.words('english')
    # remove stock market tickers like $GE
    tweet = re.sub(r'\$\w*', '', tweet)
    # remove old style retweet text "RT"
    tweet = re.sub(r'^RT[\s]+', '', tweet)
    # remove hyperlinks
    #tweet = re.sub(r'https?:\/\/.*[\r\n]*', '', tweet)
    tweet = re.sub(r'https?://[^\s\n\r]+', '', tweet)
    # remove hashtags
    # only removing the hash # sign from the word
    tweet = re.sub(r'#', '', tweet)
    # tokenize tweets
    tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True,
                               reduce_len=True)
    tweet_tokens = tokenizer.tokenize(tweet)
```

```
    tweets_clean = []
    for word in tweet_tokens:
        if (word not in stopwords_english and  # remove stopwords
                word not in string.punctuation):  # remove punctuation
            # tweets_clean.append(word)
            stem_word = stemmer.stem(word)  # stemming word
            tweets_clean.append(stem_word)

    return tweets_clean
```

[ ]:
```
def test_lookup(func):
    freqs = {('sad', 0): 4,
             ('happy', 1): 12,
             ('oppressed', 0): 7}
    word = 'happy'
    label = 1
    if func(freqs, word, label) == 12:
        return 'SUCCESS!!'
    return 'Failed Sanity Check!'
```

[ ]:
```
def lookup(freqs, word, label):
    '''
    Input:
        freqs: a dictionary with the frequency of each pair (or tuple)
        word: the word to look up
        label: the label corresponding to the word
    Output:
        n: the number of times the word with its corresponding label appears.
    '''
    n = 0  # freqs.get((word, label), 0)

    pair = (word, label)
    if (pair in freqs):
        n = freqs[pair]

    return n
```

From: https://matplotlib.org/3.1.1/gallery/statistics/confidence_ellipse.html#sphx-glr-gallery-statistics-confidence-ellipse-py

[ ]:
```
def confidence_ellipse(x, y, ax, n_std=3.0, facecolor='none', **kwargs):
    """
    Create a plot of the covariance confidence ellipse of `x` and `y`

    Parameters
    ----------
    x, y : array_like, shape (n, )
```

```
    Input data.

ax : matplotlib.axes.Axes
    The axes object to draw the ellipse into.

n_std : float
    The number of standard deviations to determine the ellipse's radiuses.

Returns
-------
matplotlib.patches.Ellipse

Other parameters
----------------
kwargs : `~matplotlib.patches.Patch` properties
"""
if x.size != y.size:
    raise ValueError("x and y must be the same size")

cov = np.cov(x, y)
pearson = cov[0, 1] / np.sqrt(cov[0, 0] * cov[1, 1])
# Using a special case to obtain the eigenvalues of this
# two-dimensionl dataset.
ell_radius_x = np.sqrt(1 + pearson)
ell_radius_y = np.sqrt(1 - pearson)
ellipse = Ellipse((0, 0),
                  width=ell_radius_x * 2,
                  height=ell_radius_y * 2,
                  facecolor=facecolor,
                  **kwargs)

# Calculating the stdandard deviation of x from
# the squareroot of the variance and multiplying
# with the given number of standard deviations.
scale_x = np.sqrt(cov[0, 0]) * n_std
mean_x = np.mean(x)

# calculating the stdandard deviation of y ...
scale_y = np.sqrt(cov[1, 1]) * n_std
mean_y = np.mean(y)

transf = transforms.Affine2D() \
    .rotate_deg(45) \
    .scale(scale_x, scale_y) \
    .translate(mean_x, mean_y)

ellipse.set_transform(transf + ax.transData)
```

```
    return ax.add_patch(ellipse)
```