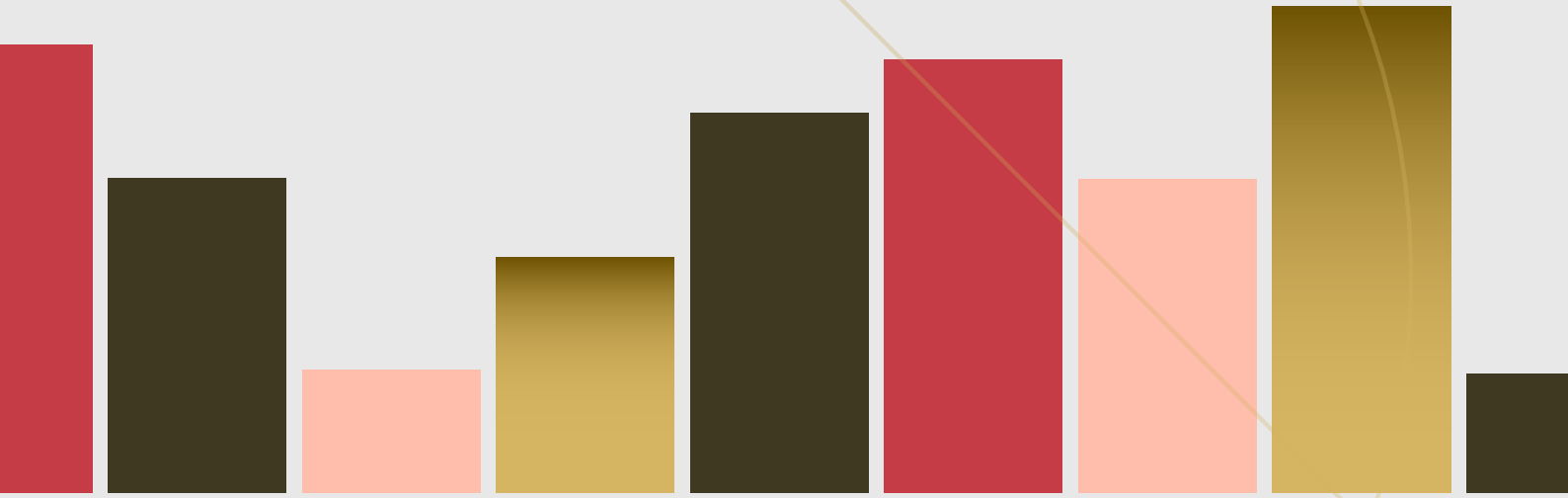
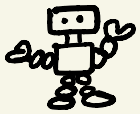
# Natural Language Processing

deeplearning.ai



Start date: 09/03

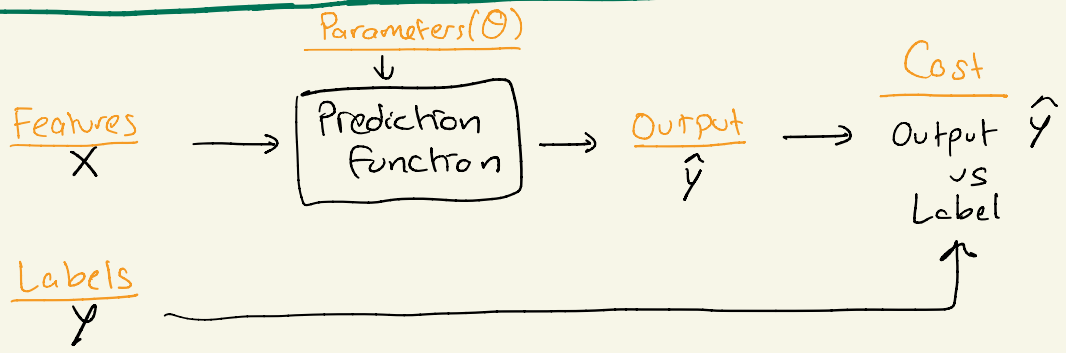
# Natural Language Processing



COURSE 1: NLP with Classification and Vector Spaces

## Week 1

### Sentiment Analysis with Logistic Regression




- Logistic regression - sentiment analysis → Positive: 1 (Obviously) / Negative: 0
  - 1) Process raw tweets
    2) Extract useful features
    3) Train logistic regression classifier (minimize the cost)

- Represent a text as a vector (in dimension "V": vocabulary)
  - SLOW
  - 1) Build a vocabulary (To encode any text as an array of numbers)
    2) Vocabulary, V ⇒ The list of unique words from all text (tweets in this case)
    3) Feature extraction: Assigning 1 and 0 depending on the appearance of every word from the vocabulary above appears in the tweet
       - It appears: Assign 1 to that feature
       - If it not: Assign 0 to that feature
       - Sparse representation: a small relative number of non-zero values
       - This representation of the tweet will have a number of features equal to the size of the entire vocabulary above.

- With this representation, a logistic regression model will have to learn n+1 parameters

4) Generate counts ➡ features into the logistic regression classifier
   - ↳ the number of times that a word appears in +/− class

- Get the positive frequency in any word in the vocabulary, count the times as it appears in the positive tweets.
  - ↳ Same thing applies to negative frequency.

freqs: dictionary mapping from (word, class) to frequency

$$X_m = [1, \sum_w freqs(w,1), \sum_w freqs(w,0)]$$

- $X_m$: features of tweet m
- 1: bias
- $\sum_w freqs(w,1)$: sum of positive frequencies
- $\sum_w freqs(w,0)$: sum of negative frequencies

⇒ Represents a tweet as a vector of three

**Preprocessing steps in NLP**
- Tokenizing the string
- Lowercasing
- Removing stop words & punctuation
- Stemming

## Preprocessing

• Stemming • & • Stop words •

To process a tweet:

Stop words

1) Remove all the words that don't add meaning to the tweets
   * Stop words and punctuation marks
2) Every word that also appears in stop words and punc. marks lists should be eliminated. → Sometimes punctuations shouldn't be eliminated, evaluate case-by-case basis

3) Stemming: transforming any word to its base stem
   the set of characters that are used to construct the word and its derivatives

tuning → tun (stem) → tune, tuned, tuning

4) To reduce vocabulary even further, lowercase every one of the words.

## Generating matrix

After extracting features using a frequencies dictionary mapping ⇒ matrix X: m rows, 3 columns: Every row would contain the features for each one of the tweets

$$X = \begin{bmatrix} 1 & X_1^{(1)} & X_2^{(1)} \\ 1 & X_1^{(2)} & X_2^{(2)} \\ \vdots & \vdots & \vdots \\ 1 & X_1^{(m)} & X_2^{(m)} \end{bmatrix}$$

```
freqs = build_freqs(tweets, labels) ⇒ build frequencies dictionary
X = np.zeros((m, 3)) ⇒ initialize matrix X
for i in range(m):
    p_tweet = process_tweet(tweets[i]) ⇒ stemming, deleting stop words, URLs, handles, and lower casing
    X[i, :] = extract_features(p_tweet, freqs)
```

- NLTK package (natural language toolkit) (nltk.org) ⇒
  - tokenize and tag
  - stemming
  - parsing → display a parse tree
  - semantic reasoning
  - identifying name entities

```
import nltk
from nltk.corpus import twitter_samples   # sample Twitter dataset from NLTK
                                          #  ↳ 5000 + & 5000 - tweets
import matplotlib.pyplot as plt
import random                             # pseudo-random number generator
nltk.download('twitter_samples')
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')
print('Number of positive tweets:', len(all_positive_tweets))
print('Number of negative tweets:', len(all_negative_tweets))
print('\nThe type of all_positive_tweets is:', type(all_positive_tweets))
print('The type of a tweet entry is:', type(all_negative_tweets[0]))
# data is stored in a list, individual tweets are stored as strings.
```

pie chart for visualizing the dataset

```
fig = plt.figure(figsize=(5,5))
labels = 'Positives', 'Negatives'
sizes = [len(all_positive_tweets), len(all_negative_tweets)] # sizes for each slide
plt.pie(sizes, labels=labels, autopct='%1.1f%%', shadow=True,
        startangle=90)
plt.axis('equal')
plt.show()
print('\033[92m' + all_positive_tweets[random.randint(0,5000)])
                  green
print('\033[91m' + all_negative_tweets[random.randint(0,5000)])
                  red
nltk.download('stopwords')
import re                                 # library for regular expression operations
import string                             # for string operations
from nltk.corpus import stopwords         # module for stopwords
from nltk.stem import PorterStemmer       # module for stemming
from nltk.tokenize import TweetTokenizer  # module for tokenizing strings
```

**Removing hashtags, retweet marks, hyperlinks**

```
tweet = all_positive_tweets[2277]  # choosing a sample
tweet2 = re.sub(r'^RT[\s]+', '', tweet)  # remove 'RT'
tweet2 = re.sub(r'https?://.*[\r\n]*', '', tweet2)
    ↳ # remove hyperlinks
tweet2 = re.sub(r'#', '', tweet2)
    ↳ # remove only hashtag sign
print(tweet2)
```

## Tokenizing the string • tokenize module – NLTK •

**tokenize:** split the strings into individual words (substrings) without blanks or tabs

```
tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True, reduce_len=True)
```

↳ #instantiate tokenizer class

- By default, it is set to True.
- If set to False, the tokenizer will downcase everything.

```
tweet_tokens = tokenizer.tokenize(tweet2)
print()
print('Tokenized string:')
print(tweet_tokens)
```

## Remove stop words and punctuations

```
stopwords_english = stopwords.words('english')  # import the english stop words list from NLTK
print('Stop words \n')
print(stopwords_english)
print('\n Punctuation \n')
print(string.punctuation)
```

### Clean up

```
tweets_clean = []
for word in tweet_tokens:  # go through every word in the list
    if (word not in stopwords_english and  # removing stopwords
        word not in string.punctuation):  # removing punctuations
        tweets_clean.append(word)
print('removed:')
print(tweets_clean)  # let's see what our cleaned tweet looks like
```

### Stemming

→ NLTK has different modules for stemming. **Porter Stemmer module** uses Martin Porter's algorithm

```
stemmer = PorterStemmer()  # instantiate stemming class
tweets_sten = []  # create an empty list to store the stems
for word in tweets_clean:
    sten_word = stemmer.sten(word)  # stemming word
    tweets_sten.append(sten_word)  # append to the list
print("stemmed words:")
print('tweets_sten')
```

# Building and Visualizing Word Frequencies

```
import nltk
from nltk.corpus import twitter_samples
import matplotlib.pyplot as plt
import numpy as np
nltk.download('stopwords')
from utils import process_tweet, build_freqs
```

**2 helper functions:**
- `process_tweet()` ↳ clean, tokenize, removes stopwords, stemming
- `build_freqs()` ↳ Counts how often a word in the corpus was associated with positive label '1' and negative label '0'.

```
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')
tweets = all_positive_tweets + all_negative_tweets  #concatenate
print("Number of tweets:", len(tweets))
```

⊛ `labels = np.append(np.ones((len(all_positive_tweets))), np.zeros((len(all_negative_tweets))))`

↳ # numpy array representing labels of the tweets

**word frequency dictionary:**

```
def build_freqs(tweets, ys):
```

↳ an mx1 array with the sentiment label of each tweet

```
    yslist = np.squeeze(ys).tolist()
```

↳ # converting np array into list because 'zip' function needs an iterable

# squeeze is necessary otherwise the list ends up with one element.

# if ys is already a list, no need to do this.

```
    freqs = {}  # start with empty dictionary
    for y, tweet in zip(yslist, tweets):
        for word in process_tweet(tweet):
            pair = (word, y)
```

alternative: `freqs[pair] = freqs.get(pair, 0) + 1`

```
            if pair in freqs:
                freqs[pair] += 1
            else:
                freqs[pair] = 1
    return freqs
```

# Each key is a 2-element tuple containing a (word, y) pair:

- **word**: an element in a processed tweet
- **y**: an integer representing the corpus ⇒ 1 for positive, 0 for negative
- **value** associated with a key is the number of times that word appears in the specified corpus

`('shame', 0.0): 19`

**Reminder of <dictionaries> in Python**

**Dictionaries:**
- Mutable } collection
- Indexed }
- Stores items as key-value pairs and uses hash tables underneath to allow lookups.
- Declared using curly brackets { }
- dictionary = {'key1': 1, 'key2': 2}

**Adding a new entry:**
- With square brackets []
- dictionary['key3'] = -5

**Accessing values and lookup keys:**
- 2 methods:
  1) square brackets
     `print(dictionary['key2'])`
     * Works if the lookup key is in the dictionary
  2) get()
     * Allows to set a default value if the key does not exist
     `print(dictionary.get('key7', -1))`

```
freqs = build_freqs(tweets, labels)
print(f'type(freqs) = {type(freqs)}') # data type
print(f'len(freqs) = {len(freqs)}') # length of the dictionary
```

## Table of word counts

```
keys = ['  ', '  ', '  ']  # select words to appear in the report.
data = []  # list representing our table of word counts
```

each element has a sublist: [<word>, <pos_count>, <neg_count>]

```
for word in keys:
    pos = 0  # initialize + and - counts
    neg = 0
    if (word, 1) in freqs:
        pos = freqs[(word, 1)]  # retrieve number of positive counts
    if (word, 0) in freqs:
        negative = freqs[(word, 0)]  # retrieve number of negative counts
    data.append([word, pos, neg])  # append the word counts to the table

data
```

## Visualizing with scatter plot

#wide discrepancies between positive and negative values ⇒ logarithmic scale rather than raw counts

```
fig, ax = plt.subplots(figsize = (8, 8))
x = np.log([x[1]+1 for x in data])  # adding 1 is to avoid log[0]
y = np.log([x[2]+1 for x in data])
ax.scatter(x,y)
plt.xlabel("Log positive count")
plt.ylabel("Log negative count")
for i in range(0, len(data)):
    ax.annotate(data[i][0], (x[i],y[i]), fontsize=12)
ax.plot([0,9], [0,9], color='red')
plt.show()
```

# Logistic Regression

- Sigmoid function which outputs a probability between zero and one

$$h(x^{(i)}, \theta) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$

- For classification:
  - Threshold = 0.5
  - ↳ Dot product between $\theta^T$ and X equals to zero at this value
    - $\theta^T x^{(i)}$

code:

```
z: input (scalar/array)
h = 1 / (1 + np.exp(-z))
```

## Training LR

- $\theta$
- $h = h(X, \theta)$
- $\nabla = \frac{1}{m} X^T (h - y)$
- $\theta = \theta - a\nabla$
- $J(\theta)$

- Goes until to a step-parameter or maximum number of iterations.

- Initialize parameters
- Classify/predict
- Get gradient
- Update
- Get loss

Gradient descent

## Testing LR

$$pred = h(X_{val}\theta) \geq 0.5 \Rightarrow \begin{bmatrix} 0.3 \\ 0.8 \\ 0.5 \\ \vdots \\ him \end{bmatrix} \geq 0.5 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ \vdots \\ predM \end{bmatrix}$$

$$\sum_{i=1}^{m} \frac{(pred^{(i)} == y_{val}^{(i)})}{m} \Rightarrow \begin{bmatrix} 0 \\ 1 \\ \vdots \\ predm \end{bmatrix} == \begin{bmatrix} 0 \\ 0 \\ \vdots \\ Yvalm \end{bmatrix} \Rightarrow \begin{bmatrix} 1 \\ 0 \\ \vdots \\ pred_m == Yvalm \end{bmatrix}$$

↳ **accuracy**: an estimate of the times that logistic regression will correctly work on unseen data

# Cost Function for Logistic Regression

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h(x^{(i)},\theta) + (1-y^{(i)})\log(1-h(x^{(i)},\theta))\right]$$

number of the training examples

- $\langle - \rangle$ ensures that cost will always be a positive number

| $y^{(i)}$ | $h(x^{(i)},\theta)$ | result vector |
|---|---|---|
| 0 | any | 0 |
| 1 | 0.99 | ~0 |
| 1 | ~0 | -inf |

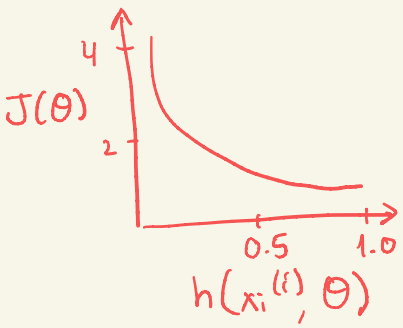
(*) this term is relevant if the label is 1. When prediction is close to the label value, loss is small. If they disagree, the overall cost goes up.

$$(1-y^{(i)})\log(1-h(x^{(i)},\theta)) \rightarrow$$

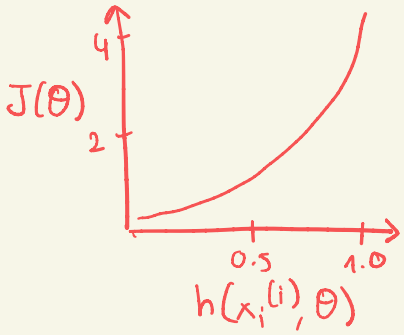
| $y^{(i)}$ | $h(x^{(i)},\theta)$ | result vector |
|---|---|---|
| 1 | any | 0 |
| 1 | 0.01 | ~0 |
| 0 | ~1 | -inf |

(*) this term is relevant when the label is 0.

$\log h(x^{(i)},\theta)$




$\log(1-h(x^{(i)},\theta))$



# Visualizing tweets and logistic regression model

```
import nltk
from os import getcwd
import pandas as pd
from nltk.corpus import twitter_samples
import matplotlib.pyplot as plt
import numpy as np
from utils import process_tweet, build_freqs  # functions defined before
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')
tweets = all_positive_tweets + all_negative_tweets
labels= np.append(np.ones((len(all_positive_tweets), 1)),
                  np.zeros((len(all_negative_tweets), 1)), axis=0)
```

```
train_pos = all_positive_tweets[:4000]
train_neg = all_negative_tweets[:4000]
```

> splitting dataset into two: 1) training 2) testing

```
train_x = train_pos + train_neg
print("Number of tweets: ", len(train_x))
data=pd.read_csv('logistic_features.csv'); # pre-prepared csv containing features
data.head(10)  # printing first 10 columns
X= data[['bias', 'positive', 'negative']].values  #get numerical values
Y = data['sentiment'].values;
print(X.shape)
theta= [7e-08, 0.005239, -0.005517] #calling a pretrained LR model
```

↳ # split our feature space into 2 parts

↳ We have a 3D feature space → bias, pos. value, neg. value

If we ignore bias, we can plot each tweet on a cartesian plane.

```
fig, ax= plt.subplots(figsize=(8,8))
colors = ['red','green']
ax.scatter(X[:, 1], X[:, 2], c=[colors[int(k)] for k in Y], s=0.1)
plt.xlabel('Positive')
plt.ylabel('Negative')
```

## plot the model alongside the data (just for visualization)

# draw a gray line to show the cutoff between the pos. and neg. regions.

gray line: $z = \theta * x = 0$

↳ $z = \theta * x = 0$

$x = [1, pos, neg]$

$z(\theta, x) = \theta_0 + \theta_1 * pos + \theta_2 * neg = 0$

$neg = (-\theta_0 - \theta_1 * pos) / \theta_2$

# there will be red and green lines that point in the direction of the corresponding sentiment.

- they are calculated using perpendicular (⊥) line to the separation line
- lines should point in the same direction as the derivative of the Logit function. → $direction = pos * \theta_2 / \theta_1$

# $f(pos, neg, w) = w0 + w1 * pos + w2 * neg = 0$ } equation for the separation plane

# $s(pos, w) = (-w0 - w1 * pos) / w2$

```
def neg(theta, pos):
    return (-theta[0] - pos * theta[1]) / theta[2]
```

# $df(pos, w) = pos * w2 / w1$ → equation for the directions of the sentiment change

```
def direction(theta, pos):
    return pos * theta[2] / theta[1]

fig, ax = plt.subplots(figsize=(8,8))
colors = ['red','green']
ax.scatter(X[:,1], X[:,2], c=[colors[int(k)] for k in Y], s=0.1)
plt.xlabel("Positive")
plt.ylabel("Negative")
maxpos = np.max(X[:,1]) # represent LR model in this chart
offset = 5000
ax.plot([0, maxpos], [neg(theta, 0), neg(theta, maxpos)], color='gray')
ax.arrow(offset, neg(theta, offset), offset, direction(theta, offset),
         head_width=500, head_length=500, fc='g', ec='g')
ax.arrow(offset, neg(theta, offset), -offset, -direction(theta, offset),
         head_width=500, head_length=500, fc='r', ec='r')
plt.show()
```

# PROGRAMMING ASSIGNMENT - Week 1

```
import nltk
from os import getcwd
nltk.download('twitter_samples')
nltk.download('stopwords')
import numpy as np
import pandas as pd
from nltk.corpus import twitter_samples
from utils import process_tweet, build_freqs
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')
test_pos = all_positive_tweets[4000:]
train_pos = all_positive_tweets[:4000]
test_neg = all_negative_tweets[4000:]
train_neg = all_negative_tweets[:4000]
train_x = train_pos + train_neg
test_x = test_pos + test_neg
train_y = np.append(np.ones((len(train_pos),1)), np.zeros((len(train_neg),1)),
                    axis=0)
test_y = np.append(np.ones((len(test_pos),1)), np.zeros((len(test_neg),1)),
                   axis=0)
print("train_y.shape = " + str(train_y.shape))
print("test_y.shape = " + str(test_y.shape))
freqs = build_freqs(train_x, train_y)   => creating frequency dictionary
```

↳

```
for y, tweet in zip(ys, tweets):
    for word in process_tweet(tweet):
        pair = (word, y)
        if pair in freqs:
            freqs[pair] += 1
        else:
            freqs[pair] = 1
print("type(freqs) = " + str(type(freqs)))
print("len(freqs) = " + str(len(freqs.keys())))
```

# Part 1: Logistic regression

## Part 1.1: Sigmoid

$$h(z) = \frac{1}{1+\exp^{-z}}$$

```
def sigmoid(z):        => z: logits
    h = 1 / (1+np.exp(-z))
    return h
```

## Part 1.2: Cost function and gradient

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log\left(h\left(z(\theta)^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(z(\theta)^{(i)}\right)\right)$$

m: training examples
$y^{(i)}$: actual label of the i-th training example
$h(z(\theta)^{(i)})$: model's prediction for the i-th training example

$$\nabla_{\theta_j} J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \left(h^{(i)} - y^{(i)}\right) x_j$$

the gradient of the cost function

weight vector: $\theta$
index across all 'm' training examples: $> i$
index of the weight $\theta_j$: $j$
feature associated w/ weight $\theta_j$: $x_j$

$$\theta_j = \theta_j - a \times \nabla_{\theta_j} J(\theta)$$

```
def gradientDescent(x, y, theta, alpha, num_iters):
    """
    Input:
        x: matrix of features => m, n+1
        y: corresponding labels of matrix x => m, 1
        theta: weight vector of dimension => n+1, 1
        alpha: learning rate
        num_iters: number of iterations you want to train your model for

    Output:
        J: the final cost
        theta: final weight vector
    """
    m = x.shape[0]
    for i in range(0, num_iters):
        z = np.dot(x, theta)
        h = 1 / (1 + np.exp(-z))
        J = -(float(1)/m) * (np.dot(y.transpose(), np.log(h)) +
                             np.dot((1-y).transpose(), np.log(1-h)))
```

$$J = \frac{-1}{m} \times \left(y^T \cdot \log(h) + (1-y)^T \cdot \log(1-h)\right)$$
$$\theta = \theta - \frac{a}{m} \times \left(x^T \cdot (h-y)\right)$$

→

```
    theta = theta - ((alpha/m) * np.dot(np.transpose(x), (h-y))
J = float(J)
return J, theta
```

## Part 2: Extracting features

```
def extract_features(tweet, freqs):
    word_l = process_tweet(tweet)
    x = np.zeros((1,3))
    x[0,0] = 1
    for word in word_l:
        x[0,1] += freqs.get((word, 1.0), 0)
        x[0,2] += freqs.get((word, 0.0), 0)
    assert(x.shape == (1,3))
    return x
```

## Part 3: Training your model

Stack the features for all training examples into a matrix X

```
X = np.zeros((len(train_x), 3))
for i in range(len(train_x)):
    X[i,:] = extract_features(train_x[i], freqs)

y = train_y
J, theta = gradientDescent(X, y, np.zeros((3,1)), 1e-9, 1500)
```

## Part 4: Test your logistic regression

```
def predict_tweet(tweet, freqs, theta):
    "y_pred = sigmoid(x.theta)"
```

(3,1) vector of weights

```
    x = extract_features(tweet, freqs)
    y_pred = sigmoid(np.dot(x, theta))
    return y_pred
```

## Part 5: Check performance

```
def test_logistic_regression(test_x, test_y, freqs, theta):
    """ Inputs:
        test_x: a list of tweets
        test_y: (m, 1) vector w/ the corresponding labels
        freqs: a dictionary w/ the frequency of each pair
        theta: weight vector of dimension (3,1)
    Output:
        accuracy
    """
    y_hat = []
    for tweet in test_x:
        y_pred = predict_tweet(tweet, freqs, theta)
        if y_pred > 0.5:
            y_hat.append(1.0)
        else:
            y_hat.append(0)
    accuracy = (y_hat == np.squeeze(test_y)).sum()/len(test_x)
    return accuracy
```

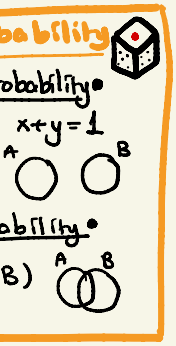
# Week 2

# Sentiment Analysis with Naïve Bayes

## Bayes' Rule

**Reminder for probability**

- complementary probability
  $P(A) = x \quad P(B) = y \quad x + y = 1$
  $1 - P(A) = P(B)$
- associated probability
  $P(A \cap B) = P(A, B)$




Conditional probabilities

- probability of B, given A happened
- looking at the elements of set A, the chance that one also belongs to set B

$$P(\text{positive} \mid \text{"happy"}) = \frac{P(\text{positive} \cap \text{"happy"})}{P(\text{"happy"})}$$

$$P(\text{"happy"} \mid \text{positive}) = \frac{P(\text{"happy"} \cap \text{positive})}{P(\text{positive})}$$

$$P(\text{positive} \mid \text{"happy"}) = P(\text{"happy"} \mid \text{positive}) \times \frac{P(\text{positive})}{P(\text{"happy"})}$$

$$P(X \mid Y) = P(Y \mid X) \times \frac{P(X)}{P(Y)}$$

Bayes' rule

## Naïve Bayes for Sentiment Analysis

**naive** : model makes the assumption that the features used for classification are all independent.

**Positive tweets corpus**

- I am happy because I am learning NLP
- I am happy, not sad.

**Negative tweets corpus**

- I am sad, I am not learning NLP
- I am sad, not happy

| word | pos | neg |
|---|---|---|
| I | 3 | 3 |
| am | 3 | 3 |
| happy | 2 | 1 |
| because | 1 | 0 |
| learning | 1 | 1 |
| NLP | 1 | 1 |
| sad | 1 | 2 |
| not | 1 | 2 |
| Total | 13 | 13 |

conditional probability ⟹

| word | pos | neg |
|---|---|---|
| I | 0.24 | 0.24 |
| am | 0.24 | 0.24 |
| happy | 0.15 | 0.08 |
| because | 0.08 | 0 |
| learning | 0.08 | 0.08 |
| NLP | 0.08 | 0.08 |
| sad | 0.08 | 0.15 |
| not | 0.08 | 0.15 |
| Sum | 1 | 1 |

$P(I \mid Pos) = 3/13$
$P(I \mid Neg) = 3/13$

$$\prod_{i=1}^{m} \frac{P(w_i \mid pos)}{P(w_i \mid neg)}$$

↳ If $> 1$:
- positive sentence

## Laplacian smoothing:

- a technique to avoid the probabilities being zero

$$P(w_i \mid class) = \frac{freq(w_i, class)}{Nclass}$$

- class $\in$ {Positive, Negative}
- Nclass = frequency of all words in class
- V = number of unique words in vocabulary

$$P(w_i \mid class) = \frac{freq(w_i, class) + 1}{Nclass + V}$$

$\Rightarrow$ 1: avoids the probability being zero
↳ it adds a new term to all the frequencies that is not correctly normalized by N class
↳ to account for this, add a new term in the denominator V.
↳ all probabilities will sum to 1

additive smoothing

### Reasons for using Laplacian:

- To avoid $P(w_i \mid class) = 0$

## Log likelihood:

- conditional probability of each word:

$$ratio(w_i) = \frac{P(w_i \mid Pos)}{P(w_i \mid Neg)}$$

↳ ratios are essential in Naïve Bayes' for binary classification

$$\prod_{i=1}^{m} \frac{P(w_i \mid pos)}{P(w_i \mid neg)} \Rightarrow \text{likelihood}$$

$$\frac{P(pos)}{P(neg)} \Rightarrow$$ prior ratio:

- taking a ratio between the negative and positive tweets
- important for unbalanced datasets - otherwise gen. 1

$$\frac{P(pos)}{P(neg)} \cdot \prod_{i=1}^{m} \frac{P(w_i \mid pos)}{P(w_i \mid neg)} \Rightarrow$$ Full Naïve Bayes' formula

Multiplication of many numbers between zero and one → risk of numerical underflow

to solve:
- use a log of the score rather than the raw score

$$\log\left(\frac{P(pos)}{P(neg)} \prod_{i=1}^{m} \frac{P(w_i \mid pos)}{P(w_i \mid neg)}\right)$$

1) $\log(a * b) = \log(a) + \log(b)$

2) $\log \frac{P(pos)}{P(neg)} + \sum_{i=1}^{m} \log \frac{P(w_i \mid pos)}{P(w_i \mid neg)}$

log prior

log likelihood
↳ $\sum_{i=1}^{m} \lambda(w_i)$

$$\underbrace{\lambda(w)}_{\text{Lambda}} = \log \frac{P(w_i \mid pos)}{P(w_i \mid neg)}$$

# Training Naive Bayes

✓ No gradient descent – Just counting the frequencies of the words in a corpus

## Steps of a supervised ML project (NLP):

**Step 1:** Collect and annotate corpus

**Step 2:** Preprocess
- Lowercase
- Remove punctuations, urls, nones
- Remove stop words
- Stemming
- Tokenize sentences

**Step 3:** 1) Word count: $freq(w, class)$

2) $P(w_i|class) \longrightarrow \frac{freq(w, class) + 1}{N_{class} + V_{class}}$

**Step 4:** Get lambda

$$\lambda(w) = \log \frac{P(w|pos)}{P(w|neg)}$$

**Step 5:** Get the log prior

$$logprior = \frac{D_{pos}\ (number\ of\ +\ tweets)}{D_{neg}\ (number\ of\ -\ tweets)}$$

⊛ If dataset is balanced, $D_{pos} = D_{neg}$ and $logprior = 0$

# Visualizing Naive Bayes

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from utils import confidence_ellipse
```

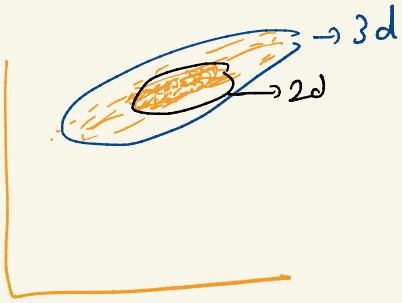
- calculate the likelihoods for each tweet
- plot the samples:

```
fig, ax = plt.subplots(figsize=(8,8))
colors = ['red', 'green']
ax.scatter(data.positive, data.negative, c=[colors[int(k)] for k in data.sentiment],
           s=0.1, marker='*')

plt.xlim(-250,0)
plt.ylim(-250,0)
plt.xlabel("Positive")
plt.ylabel("Negative")
```

- confidence ellipse:
  - * a way to visualize a 2D random variable

```
fig, ax = plt.subplots(figsize=(8,8))
colors = ['red', 'green']
ax.scatter(data.positive, data.negative, c=[colors[int(k)] for k in data.sentiment],
           s=0.1, marker='*')

plt.xlim(-200,40)
plt.ylim(-200,40)
plt.xlabel("Positive")
plt.ylabel("Negative")
data_pos = data[data.sentiment==1]
data_neg = data[data.sentiment==0]
```

precalculated likelihoods




2 sd:
```
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')
```

3 d:
```
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()
plt.show()
```

## Testing Naive Bayes

Tweet: I passed the NLP interview

↳ Sum of the lambda score of each word + log prior $> 0$ = positive sentiment

$X_{val}$, $Y_{val}$, $\lambda$, log prior

↳ score = predict ($X_{val}$, $\lambda$, log prior)

pred = score $> 0$

$\frac{1}{m} \sum_{i=1} (pred_i == y_{val_i})$ ⇒ accuracy

## Applications of Naive Bayes

**Reminder:**

$P(pos|tweet) \propto P(pos)\, P(tweet|pos)$

$P(neg|tweet) \propto P(neg)\, P(tweet|neg)$

$$\frac{P(pos|tweet)}{P(neg|tweet)} = \frac{P(pos)}{P(neg)} \prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)}$$

- author identification
- spam detection
- information retrieval
- word disambiguation

- One of the earliest use of Naive Bayes was filtering relevant and irrelevant documents in a database. ⇒ We only need to calculate likelihood (no prior knowledge of whether the doc is irr. or rel.)

Information retrieval:

$$P(document_K|query) \propto \prod_{i=0}^{|query|} P(query_i|document_K)$$

retrieve document if $P(document_K|query) >$ threshold

## Naïve Bayes Assumptions

1) Independence

- Assume inpendence between the predictors or features associated with each class
  ↳ Words in a text are independent of one another
    ↳ Under/over estimates the conditional probabilities of individual words

2) Relative frequency in corpus

- Naive Bayes relies on the distribution of the training dataset

## Error Analysis

Causes of errors:
1) Semantic meaning loss in the pre-processing step
   Due to ↳ Removing punctuation and stop words
2) Word order affecting the meaning of the sentence
3) Quirks of human language
   ↳ Adversarial attacks

## Week 2: Assignment

```
def count_tweets (result, tweets, ys):
    " Input:
        result: a dictionary that will be used to map each pair to its frequency
        tweets: a list of tweets
        ys: a list corresponding to the sentiment of each tweet (either 0 or 1)
      Output:
        result: a dictionary mapping each pair to its frequency
    "
    pair = (word, y)
    if pair in result:
        result[pair] += 1
    else:
        result[pair] = 1
return result
```

output: {('happi', 1): 1, ('trick', 0): 1}

freqs = count_tweets({}, train_x, train_y) ⇒ building freqs dictionary

```
def train_naive_bayes (freqs, train_x, train_y):
    loglikelihood = {}
    logprior = 0
    vocab = set([pair[0] for pair in freqs.keys()])
    V = len(vocab)
    N_pos = N_neg = 0
    for pair in freqs.keys():
        if pair[1] > 0:
            N_pos += 1
        else:
            N_neg += 0
```

```
<indent> D = len(train_y)
        D_pos = (len(list(filter(lambda x: x > 0, train_y))
        D_neg = (len(list(filter(lambda x: x <= 0, train_y))
        logprior = np.log(D_pos) - np.log(D_neg)
        for word in vocab:
            freq_pos = lookup(freqs, word, 1)
            freq_neg = lookup(freqs, word, 0)
            p_w_pos = (freq_pos + 1) / (N_pos + V)
            p_w_neg = (freq_neg + 1) (N_neg + V)
            loglikelihood[word] = np.log(p_w_pos / p_w_neg)
        return logprior, loglikelihood
logprior, loglikelihood = train_naive_bayes(freqs, train_x, train_y)
def naive_bayes_predict(tweet, logprior, loglikelihood):
    word_l = process_tweet(tweet)
    p = 0
    p += logprior
    for word in word_l:
        if word in loglikelihood:
            p += loglikelihood[word]
    return
def test_naive_bayes(test_x, test_y, logprior, loglikelihood):
    accuracy = 0
    y_hats = []
    for tweet in test_x:
        if naive_bayes_predict(tweet, logprior, loglikelihood) > 0:
            y_hat_i = 1
        else:
            y_hat_i = 0
    error = np.mean(np.absolute(y_hats - test_y))
    accuracy = 1 - error
    return
```

$$p = \text{logprior} + \sum_{i}^{N} (\text{loglikelihood}_i)$$

```
def get_ratio(freqs, word):
    pos_neg_ratio = {'positive': 0, 'negative': 0, 'ratio': 0.0}
    pos_neg_ratio['positive'] = lookup(freqs, word, 1)
    pos_neg_ratio['negative'] = lookup(freqs, word, 0)
    pos_neg_ratio['ratio'] = ((pos_neg_ratio['positive'] + 1) / (pos_neg_ratio['negative'] + 1))
```

→ defining the level of positiveness or negativeness

```
def get_words_by_threshold(freqs, label, threshold):
    word_list = {}
    for key in freqs.keys():
        word, _ = key
        pos_neg_ratio = get_ratio(freqs, word)
        if label == 1 and pos_neg_ratio >= threshold:
            word_list[word] = pos_neg_ratio
        elif label == 0 and pos_neg_ratio <= threshold:
            word_list[word] = pos_neg_ratio
    return word_list
```

```
get_words_by_threshold(freqs, label=  , threshold=  )
```

0 for neg, 1 for pos ↑ (label)

threshold value ↑ (threshold)

# Week 3

## Vector Space Models