**Course 1: Natural Language Processing with Classification and Vector Spaces**

* In course one, you will learn to distinguish between pieces of text with positive sentiments and negative sentiments.
  + You will do so using logistic regression and Naïve Bayes Classifiers.
* You will also learn to represent words, queries and documents including other pieces of text as numbers in vectors.
* You will build your first machine translation system and you will learn about locality sensitive hashing, which is a method that will help you with efficient search.

**Supervised ML & Sentiment Analysis**

**Supervised ML (training)**

* In supervised machine learning, you have **input features *X*** and a set of **labels *Y***.
* To make sure you’re getting the most accurate predictions based on your data, your goal is to minimize **error rates** or **cost** as much as possible.
* To do this, you’re going to run the **prediction function** which takes in parameters θ to map **features** to **output labels** ***Ŷ***.
* The best mapping from features to labels is achieved when the difference between the **expected values *Y*** and the **predicted values** ***Ŷ*** is minimized, which the **cost function** does by comparing your output ***Ŷ*** is to your label ***Y***.
* Then, you can update your parameters and repeat the whole process until your cost is minimized.

Diagram

Description automatically generated

**Sentiment analysis**

* **Example Tweet**: I am happy because I am learning NLP
* **Objective**: Predict whether a tweet has a positive or negative sentiment.
* You will do this by starting with a **training set**, where tweets with a **positive sentiment** have a label of **1**, and tweets with a **negative sentiment** have a label of **0**.
* For this task, you will be using **logistic regression classifier**, which assigns its observations to two distinct classes.
* To get started building a **logistic regression classifier** that is capable of predicting sentiments of an arbitrary tweet, you will first process the raw tweets in your training sets and extract useful features. Then, you will train your logistic regression classifier while minimizing the cost. And finally, you’ll be able to make your predictions.
* Steps:

1. Extract the features
2. Train the model
3. Classify the tweet based on the trained model

**Vocabulary & Feature Extraction**

* In order to represent a text as a vector, you first have to build a **vocabulary**, and that will allow you to encode any text or any tweet as an array of numbers.

**Vocabulary**

* Your **Vocabulary *V*** would be the list of unique words from your list of tweets. To get that list, you’ll have to go through all the words from all your tweets and save every new word that appears in your search.
* Tweets: [tweet\_1, tweet\_2, …, tweet\_m] 🡪 [I am happy because I am learning NLP, …, …, I hated the movie]
* *V =* [I, am, happy, because, learning, NLP, …, hated, the, movie]
* Note that the word “I” and the word “am” would not be repeated in the vocabulary.

**Feature extraction**

* In order to extract features from a tweet using the vocabulary, you’d have to check if every word from your vocabulary appears in the tweet. If it does, you would assign a value of 1 to that feature. If it doesn’t appear, you’d assign a value of 0.
* Example Tweet: I am happy because I am learning NLP
* [I, am, happy, because, learning, NLP, …, hated, the, movie]
* [1, 1, 1, 1, 1, 1, …, 0, 0, 0]
* In this example, the representation of your tweet would have six 1s and many 0s. These correspond to every unique word from your vocabulary that isn’t in the tweet.
* This type of representation with a small relative number of non-zero values is called a **sparse representation**.

**Problems with sparse representations**

* With the sparse representation, a logistic regression model would have to learn **n + 1** parameters, where n would be equal to the size of your vocabulary and you can imagine that for large vocabulary sizes, this would be problematic.

1. Large training time
2. Large prediction time

* I am happy because I am learning NLP
* [1, 1, 1, 1, 1, 1, …, 0, 0, 0]
* [θ0, θ1, θ2, …, θn]
* n = |*V*|

**Negative and Positive Frequencies**

* You can generate **counts** and use them as **features** into your **logistic regression classifier**.
  + Specifically, given a word, you want to keep track of the number of times that word shows up in the positive class. And given another word, you want to keep track of the number of times that word showed up in the negative class.
  + Using both those counts, you can then extract features and use those features into your logistic regression classifier.

**Positive and Negative Counts**

* Corpus:
  + I am happy because I am learning NLP
  + I am happy
  + I am sad, I am not learning NLP
  + I am sad
* Vocabulary:
  + I, am, happy, because, learning, NLP, sad, not
* Two classes: Positive Sentiment & Negative Sentiment
  + Positive Tweets:
    - I am happy because I am learning NLP
    - I am happy
  + Negative Tweets:
    - I am sad, I am not learning NLP
    - I am sad
* Word frequencies in classes:

|  |  |  |
| --- | --- | --- |
| **Vocabulary** | **PosFreq (1)** | **NegFreq (0)** |
| I | 3 | 3 |
| am | 3 | 3 |
| happy | 2 | 0 |
| because | 1 | 0 |
| learning | 1 | 1 |
| NLP | 1 | 1 |
| sad | 0 | 2 |
| not | 0 | 1 |

* In practice, when coding, this table is a **dictionary mapping** from (word, class) to frequency. So, it maps the word and its corresponding class to the frequency or the number of times that word showed up in the class.

**Feature Extraction with Frequencies**

* You will now learn to encode a tweet or specifically, represent it as a vector of dimension three.
  + In doing so, you’ll have a much faster speed for your logistic classifier, because instead of learning *V* features, you only have to learn three features.

**Feature extraction**

* *freqs*: dictionary mapping from (word, class) to frequency
* Now that you’ve built your frequencies dictionary, you can use it to extract useful features for sentiment analysis. What does a feature look like? Let’s look at an arbitrary tweet m.
* *Xm* = [1, ,
  + *Xm* 🡪 Features of tweet m
  + 1 🡪 Bias
  + 🡪 Sum of Positive Frequencies
  + 🡪 Sum of Negative Frequencies
* Example:
  + I am sad, I am not learning NLP

|  |  |  |
| --- | --- | --- |
| **Vocabulary** | **PosFreq (1)** | **NegFreq (0)** |
| I | 3 | 3 |
| am | 3 | 3 |
| happy | 2 | 0 |
| because | 1 | 0 |
| learning | 1 | 1 |
| NLP | 1 | 1 |
| sad | 0 | 2 |
| not | 0 | 1 |

* + = 3 + 3 + 1 + 1 + 0 + 0 = 8
  + = 3 + 3 + 1 + 1 + 2 + 1 = 11
  + Therefore, the representation of this tweet would be equal to the vector:
    - *Xm* = [1, 8, 11]

**Preprocessing**

* Two major concepts of preprocessing:

1. **Stemming**
2. **Stop words**

**Preprocessing: stop words and punctuation**

* Example Tweet:
  + @YMourri and @AndrewYNg are tuning a GREAT AI model at https://deeplearning.ai!!!
* **Stop words** 🡪 and, is, are, at, has, for, a
* **Punctuation** 🡪 , . : ! “ ‘
* Every word from the tweet that also appears on the list of stop words should be eliminated. So, you’d have eliminate the words: and, are, a, at.
* There are only exclamation points in this example.
* The tweets without stop words and punctuation:
  + @YMourri @AndrewYNg tuning GREAT AI model https://deeplearning.ai
* However, note that in some contexts, you won’t have to eliminate punctuation. So, you should think carefully about whether punctuation adds important information to your specific NLP task or not.

**Preprocessing: Handles and URLs**

* Tweets and other types of texts often have **handles** and **URLs**, but these don’t add any value for the task of sentiment analysis.
* Tweet with Handles and URLs eliminated:
  + tuning GREAT AI model
  + At the end of this process, the resulting tweet contains all the important information related to its sentiment. It is clearly a positive tweet, and a sufficiently good model should be able to classify it.

**Preprocessing: Stemming and lowercasing**

* **Stemming** in NLP is simply transforming any word to its base stem, which you could define as the set of characters that are used to construct the word and its derivatives.
  + tuning, tune, tuned 🡪 tun
  + Your vocabulary would be significantly reduced when you perform this process for every word in the corpus.
* To reduce your vocabulary even further without losing valuable information, you’d have to **lowercase** every one of your words.
  + GREAT, Great, great 🡪 great
* The final preprocessed tweet as a list of words: [tun, great, ai, model]

**Preprocessing Recap**

* When preprocessing, you have to perform the following:

1. Eliminate handles and URLs
2. Tokenize the string into words.
3. Remove stop words like "and, is, a, on, etc."
4. Stemming- or convert every word to its stem. Like dancer, dancing, danced, becomes 'danc'. You can use **porter stemmer** to take care of this.
5. Convert all your words to lower case.

**Putting it All Together**

**General overview**

* Example Tweet: I am Happy Because i am learning NLP @deeplearning
* **Preprocessing** 🡪 [happy, learn, nlp]
* With that list of words, you would be able to get a nice representation using a **frequency dictionary mapping**, and finally, get a **vector** with a **bias unit**, and two additional features that store the **sum positive frequencies** and the **sum negative frequencies** 🡪 [1, 4, 2]
* In practice, you would have to perform this process on a set of m tweets. So, given a set of multiple raw tweets, you would have to preprocess them one by one to get these sets of lists of words one for each of your tweets. And finally, you’d be able to extract features using a frequencies dictionary mapping.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| I am Happy Because i am learning NLP @deeplearning | 🡪 | [happy, learn, nlp] | 🡪 | [[1, 40, 20,], |
| I am sad not learning NLP | 🡪 | [sad, not, learn, nlp] | 🡪 | [1, 20, 50], |
| … | 🡪 | … | 🡪 | … |
| I am sad :( | 🡪 | [sad] | 🡪 | [1, 5, 35]] |

* At the end, you would have a matrix **X** with m rows and three columns where every row would contain the features for each one of your tweets.

**General Implementation**

* The general implementation of this process is rather easy. First, you build the frequencies dictionary.
* Then, initialize the matrix X to match your number of tweets.
* After that, you’ll want to go over through your sets of tweets carefully, deleting stop words, stemming, deleting URLs and handles, and lower casing.
* And finally, extract the features by summing up the positive and negative frequencies of the tweets.

Graphical user interface, text, application

Description automatically generated

**Logistic Regression Overview**

* **Logistic regression** makes use of a **sigmoid function** which outputs a probability between zero and one.

**Overview of Logistic Regression**

* Chart

  Description automatically generatedThe function used to classify in logistic regression is the **sigmoid function**. It depends on the parameters θ and the features vector *x*(i), where “i” is used to denote the *i*th observation or data point.
* For classification, a threshold is needed. Usually, it is set to be 0.5.
  + So, whenever the dot product is equal or greater than zero, the prediction is positive.
  + Whenever the dot product is less than zero, the prediction is negative.
* Example Tweet: @YMourri and @AndrewYNg are tuning a GREAT AI model
* After preprocessing 🡪 [tun, ai, great, model]
* After extracting features 🡪
* Chart, line chart

  Description automatically generatedAssuming that you already have an optimum sets of parameters θ, you would be able to get the value of the sigmoid function, in this case equal to 4.92, and finally predict a positive sentiment.

**Logistic Regression: Training**

**Training LR**

* To train your logistic regression classifier, iterate until you find the set of parameters θ that minimizes your cost function.
* Let us suppose that your loss only depends on the parameters θ1 and θ2. Chart

  Description automatically generated
* Let’s look at this process in more detail:
  + You initialize your parameter *θ*, that you can use in your sigmoid, you then compute the **gradient** that you will use to update *θ*, and then calculate the cost. You keep doing so until good enough.

Diagram

Description automatically generated

* This algorithm is known as **gradient descent**.

**Logistic Regression: Testing**

**Testing logistic regression**

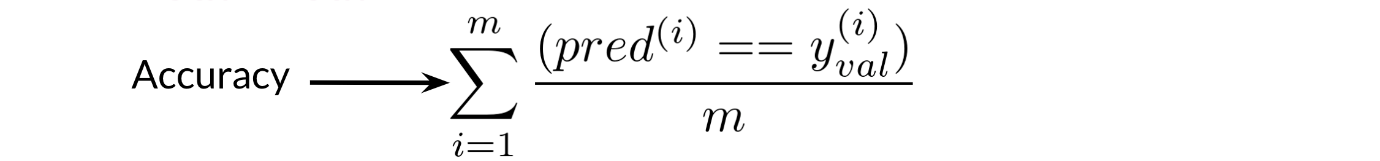
* *X*val, *Y*val 🡪 **Validation set**; data that was set aside during trainings
* *θ* 🡪 the sets of **optimum parameters** that you got from training on your data
* h(*X*val, *θ*) 🡪 **sigmoid function**
  + A picture containing chart

    Description automatically generatedpred = h(*X*val, *θ*) ≥ 0.5 🡪 You will evaluate if each value of h of θ is greater than or equal to a threshold value, often set to 0.5.

A picture containing diagram

Description automatically generated

* At the end, you’ll have a vector populated with 0s and 1s indicating predicted negative and positive examples, respectively.
* After building the predictions vector, you can compute the accuracy of your model over the validation sets.
  + Diagram

    Description automatically generatedTo do so, you will compare the predictions you have made with the true value for each observation from your validation data. If the values are equal and your predictions are correct, you’ll get a value of 1, and 0 otherwise.
* After you have compared the values of every prediction with the true labels of your validation set, you can get the total times that your predictions were correct by summing up the vector of the comparisons.
* Finally, you’ll divide that number over the total number *m* of observations from your validation sets.
  + This metric (**accuracy**) gives an estimate of the times that’s your logistic regression will correctly work on unseen data.