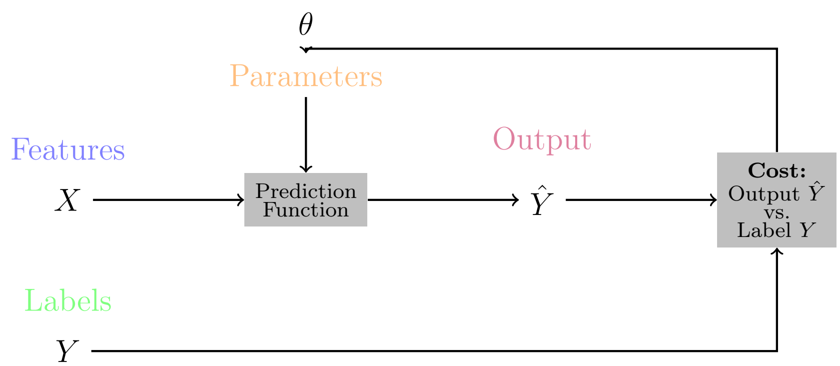
Contents

1 Classification and Vector Spaces

1.1 Supervised ML and Sentiment Analysis

In supervised ML, we have input features X and a set of labels Y. To get the most accurate predictions, we try to minimize our *error rates* or *cost function* as much as possible: to do this, we'll run our prediction function which takes in parameters θ to map you input features to output labels $\hat{Y}$. The best mapping is achieved when the difference between the expected values Y and the predicted values $\hat{Y}$ is minimized, which the cost function does by comparing how closely your output $\hat{Y}$ is to your label Y. You can then update your parameters and repeat the whole process until your cost is minimized.



How about the supervised ML classification task of sentiment analysis? Suppose we're given a tweet that says, "I'm happy because I'm learning NLP": and the objective in the task is to predict whether a tweet has a positive or negative sentiment. We'll do this by starting with a training set where tweets with a positive label have a label of unit value, and tweets with a negative sentiment have a label of zero. To get started building a logistic regression classifier that's capable of predicting sentiments of an arbitrary tweet, we first need to process the raw tweets in our training data set and extract useful features. Then, we will train our logistic regression classifier while minimimizing the cost. Finally, we'll be able to make predictions.

How to represent text as a vector In order to represent text as a vector, we need to first build a vocabulary. We define the vocabulary V as the *set* of unique words from your input data (e.g. your listing of tweets). To get this listing, we quite literally need to comb through all words from all input data and save every new word that appears in our search. To represent a tweet as a vector, we can use a one-hot encoding with our vocabulary: i.e. each tweet will be represented with a length $|V|$ vector where elements are binary-valued - a one indicates the word is in the tweet and a zero indicates the absence of a word in a tweet. We call this a *sparse* representation because the number of non-zero entries is relatively small when compared with the number of zero entries. Realize that if we are running a logistic regression, we would require learning $|V| + 1$ parameters which can be problematic for large vocabularies. If not prohibitive, it would make training models take excessive time and making predictions would be expensive.

Negative and positive frequencies Let's discuss how to generate counts which can be used as features in our logistic regression classifier. Specifically, given a word, we wish to keep track of the number of times that it shows up as the positive class. Given another word, we wish to track how many times that word shows up in the negative class. Using both these counts, we can then extract features and use those features in our logistic regression classifier. Suppose we have the following corpus of tweets:

```
I am happy because I am learning NLP
I am happy
I am sad, I am not learning NLP
I am sad
```

Then our vocabulary is given by

Vocabulary
I
am
happy
because
learning
NLP
sad
not

For this particular example of sentiment analysis, we only have two sentiments (i.e. two classes): one class is associated with a positive sentiment and the other with a negative sentiment. So, taking your corpus, you'd have a set of two tweets that belong to the positive class, and two tweets which belong to the negative class. Let's calculate the positive frequencies by examining the first two tweets:

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

The same logic applies applies to getting negative frequencies. We can calculate these by examining our last two training examples.

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

So, we can now have an entire table for our corpus, where for each entry in V we associate with it a scalar value `PosFreq(1)` and another scalar value `NegFreq(0)`. In practice, we use a Python dictionary `freqs` mapping from `(word, class)` $\rightsquigarrow$ frequency.

Feature extraction with frequencies Whereas we previously learned to encode a tweet as a vector of length $|V|$, we will now use our frequency counts to represent each tweet as a vector of length equal to one plus the number of classes in our set of labels. This gives us a much faster speed for our logistic regression classifier. How can we do this, exactly? We represent each tweet as follows:

$$\underbrace{X_m}_{\substack{\text{Features of}\\ \text{tweet } m}} = \left[\underbrace{1}_{\text{Bias}}, \underbrace{\sum_w \texttt{freqs}(w,1)}_{\substack{\text{Sum Pos.}\\ \text{Frequences}}}, \underbrace{\sum_w \texttt{freqs}(w,0)}_{\substack{\text{Sum Neg.}\\ \text{Frequencies}}}\right]$$

I.e. the first feature is a bias unit equal to unit value, the second is the sum of positive frequencies for every unique word on tweet m, and the third is the sum of negative frequencies for every unique word on the tweet. So, to extract the features for this *representation*, we only have to sum frequencies of words, which is straightforward. Let's look at an example: "I am sad, I am not learning NLP". The only words in our vocabulary that don't appear in this sentence are "happy" and "because": if we sum up the `PosFreq(1)` associated with the remaining words in our vocabulary, i.e. the words that appear in this tweet, we get a scalar value of eight. We do the same for the negative frequencies, and we get a scalar value of eleven. So, we represent "I am sad, I am not learning NLP" $\rightsquigarrow [1, 8, 11]$.

Preprocessing There are two major concepts here: stemming and "stop words". We'll learn how to apply these preprocessing steps to our data. Stop words are defined as those which don't add significant meaning to the tweets; we *might* also choose to remove punctuation (if we decide it doesn't provide information in our context). In practice, this means comparing our tweet against two sets: one with stop words (in English) and another with punctuation.

Stop Words	Punctuation
and	,
is	.
are	:
at	!
has	“
for	’
a	

In practice the list of stop words and punctuation marks are much larger, but for pedagogical purposes these will serve well. We might start out with a tweet like

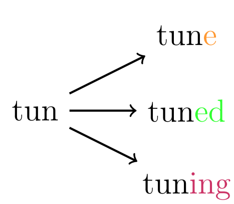
```
@YMourri and @AndrewYNg are tuning a GREAT AI Model at https://deeplearning.ai!!!
```

We then preprocess by stripping stop words "and", "are", a"at", and "a". The only punctuation that appears in this tweet that's also in our list is the exclamation point(s). We might further decide that tweets should have handles and URLs removed, because these don't add value for the specific task of sentiment analysis. In the end, we end up with a data point that looks like

```
tuning GREAT AI model
```

It's clearly a positive tweet, and a sufficiently good model should be able to classify it. Now that the tweet contains the minimum necessary information, we can perform *stemming* for every word.

Stemming 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 words and its derivatives. Let's look at the first word in the example: its stem is "tun", since



If we were to perform stemming on our entire corpus, the words "tune", "tuned", and "tuning" all get reduced to the stem "tun". So, your vocabulary would be significantly reduced in performing this process. You can further reduce the size of the vocabulary without losing valuable information by *lower-casing* every word, e.g. the words "GREAT", "Great", and "great" all get treated as the same word. Perhaps our final preprocessed tweet looks like

`[tun, great, ai, model]`

In summary, for our example of sentiment analysis on tweets, we might preprocess as follows:

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. E.g. dancer, dancing, danced, becomes "danc". You can use Porter Stemmer to take care of this.
5. Convert all words to lowercase.

As an applied example:

$$\text{I am Happy Because I am learning NLP @deeplearning} \xrightarrow{\text{Preprocessing}} [\texttt{happy, learn, nlp}] \xrightarrow{\text{Feature Extraction}} [1, 4, 2]$$

where 1 is our bias term, 4 is the sum of positive frequencies, and 2 is the sum of negative frequencies. In practice, we are given a set of m raw tweets, and so wehave to process them one-by-one to process them into an $m \times 3$ matrix, where each row describes the features for a given tweet.

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

The process is simple: (i) build the frequencies dictionary, (ii) initialize the matrix X to match the number of tweets, (iii) go through your sets of tweets and carefully preprocess by deleting stop words, stemming, deleting URLs/handles, and lowercasing, and finally (iv) extract the features by summing up the positive and negative frequencies of each of the tweets.

```
freqs = build_freqs(tweets, labels)           # Build frequencies dictionary.
X = np.zeros((m,3))                           # Initialize matrix X.
for i in range(m):                            # For every tweet:
  p_tweet = process_tweet(tweets[i])          #   Process tweet.
  X[i,:] = extract_features(p_tweet, freqs)   #   Extract features.
```