## 2.1 The Task

Your goal in this assignment is to implement a working Natural Language Processing (NLP) system, i.e., a sentiment polarity analyzer, using binary logistic regression. You will then use your algorithm to determine whether a review is positive or negative using movie reviews as data. You will do some very basic feature engineering, through which you are able to improve the learner's performance on this task. You will write two programs: `feature.{py|java|cpp|m}` and `lr.{py|java|cpp|m}` to jointly complete the task. The programs you write will be automatically graded using the Autolab system. You may write your programs in **Octave, Python, Java,** or **C++**. However, you should use the same language for all parts below.

**Note**: Before starting the programming, you should work through section 1.3 to get a good understanding of important concepts that are useful for this programming section.

## 2.2 The Datasets

**Datasets** Download the tar file from Autolab ("Download handout"). The tar file will contain all the data that you will need in order to complete this assignment. The handout contains data from the Movie Review Polarity dataset. [1] In the data files, each line is a data point that consists of a label (0 for negatives and 1 for positives) and a attribute (a set of words as a whole). The label and attribute are separated by a tab.[2] In the attribute, words are separated using white-space (punctuations are also separated with white-space). All characters are lowercased. The format of each data point (each line) is `label\tword1 word2 word3 ... wordN\n`.

Examples of the data are as follows:

```
1 david spade has a snide , sarcastic sense of humor that works ...
0 " mission to mars " is one of those annoying movies where , in ...
1 anyone who saw alan rickman's finely-realized performances in ...
1 ingredients : man with amnesia who wakes up wanted for murder , ...
1 ingredients : lost parrot trying to get home , friends synopsis : ...
1 note : some may consider portions of the following text to be ...
0 aspiring broadway composer robert ( aaron williams ) secretly ...
0 america's favorite homicidal plaything takes a wicked wife in " ...
```

We have provided you with two subsets of the movie review dataset. Each dataset is divided into a training, a validation, and a test dataset. The small dataset (`smalltrain_data.tsv`, `smallvalid_data.tsv`, and `smalltest_data.tsv`) can be used while debugging your code. We have included the reference output files for this dataset after **30 training epochs** (see directory `smalloutput/`). We have also included a larger dataset (`train_data.tsv`, `valid_data.tsv`, `test_data.tsv`) with reference outputs for this dataset after **60 training epochs** (see directory `largeoutput/`) . This dataset can be used to ensure that your code runs fast enough to pass the autograder tests. Your code should be able to perform 60-epoch training and finish predictions through all of the data in less than one minute for each of the models: one minute for Model 1 and one minute for Model 2.

**Dictionary** We also provide a dictionary file (`dict.txt`) to limit the vocabulary to be considered in this assignment (this dictionary is constructed from the training data, so it includes all the words from the training data, but some words in validation and test data may not be present in the dictionary). Each line in the dictionary file is in the following format: `word\tindex\n`. Words (column 1) and indexes (column 2) are separated with whitespace. Examples of the dictionary content are as follows:

```
films 0
adapted 1
from 2
comic 3
```

## 2.3 Model Definition

Assume you are given a dataset with $N$ training examples and $M$ features. We first write down the *negative* conditional log-likelihood of the training data in terms of the design matrix $\mathbf{X}$, the labels $\mathbf{y}$, and the parameter vector $\boldsymbol{\theta}$. This will be your objective function $J(\boldsymbol{\theta})$ for gradient descent. (Recall that $i$th row of the design matrix $\mathbf{X}$ contains the features $\mathbf{x}^{(i)}$ of the $i$th training example. The $i$th entry in the vector $\mathbf{y}$ is the label $y^{(i)}$ of the $i$th training example. Here we assume that each feature vector $\mathbf{x}^{(i)}$ contains a bias *feature*, e.g. $x_0^{(i)} = 1 \ \forall i \in \{1, \ldots, N\}$. As such, **the bias parameter is folded into our parameter vector $\boldsymbol{\theta}$.**

Taking $\mathbf{x}^{(i)}$ to be a $(M+1)$-dimensional vector where $x_0^{(i)} = 1$, the likelihood $p\left(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}\right)$ is:

$$p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}) = \prod_{i=1}^{N} p(y^{(i)}|\mathbf{x}^{(i)}, \boldsymbol{\theta}) = \prod_{i=1}^{N} \left( \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right)^{y^{(i)}} \left( \frac{1}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right)^{(1-y^{(i)})} \tag{2.1}$$

$$= \prod_{i=1}^{N} \frac{\left(e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right)^{y^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \tag{2.2}$$

Hence, the negative conditional log-likelihood is:

$$J(\boldsymbol{\theta}) = -\log p\left(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta}\right) = \sum_{i=1}^{N} -y^{(i)} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)}\right) + \log\left(1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}\right) \tag{2.3}$$

The partial derivative of the negative log-likelihood $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}_j$ , $j \in \{0, ..., M\}$ is:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = -\sum_{i=1}^{N} \mathbf{x}_j^{(i)} \left[ y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{2.4}$$

The gradient descent update rule for binary logistic regression for parameter element $\boldsymbol{\theta}_j$ is

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j - \eta \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} \tag{2.5}$$

Then, the stochastic gradient descent update for parameter element $\boldsymbol{\theta}_j$ using the $i$th datapoint $(\mathbf{x}^{(i)}, y^{(i)})$ is:

$$\boldsymbol{\theta}_j \leftarrow \boldsymbol{\theta}_j + \eta \mathbf{x}_j^{(i)} \left[ y^{(i)} - \frac{e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}}{1 + e^{\boldsymbol{\theta}^T \mathbf{x}^{(i)}}} \right] \tag{2.6}$$

## 2.4 Implementation

### 2.4.1 Overview

The implementation consists of two programs, a feature extraction program (`feature.{py|java|cpp|m}`) and a sentiment analyzer program (`lr.{py|java|cpp|m}`) using binary logistic regression. The programming pipeline is illustrated as follows.

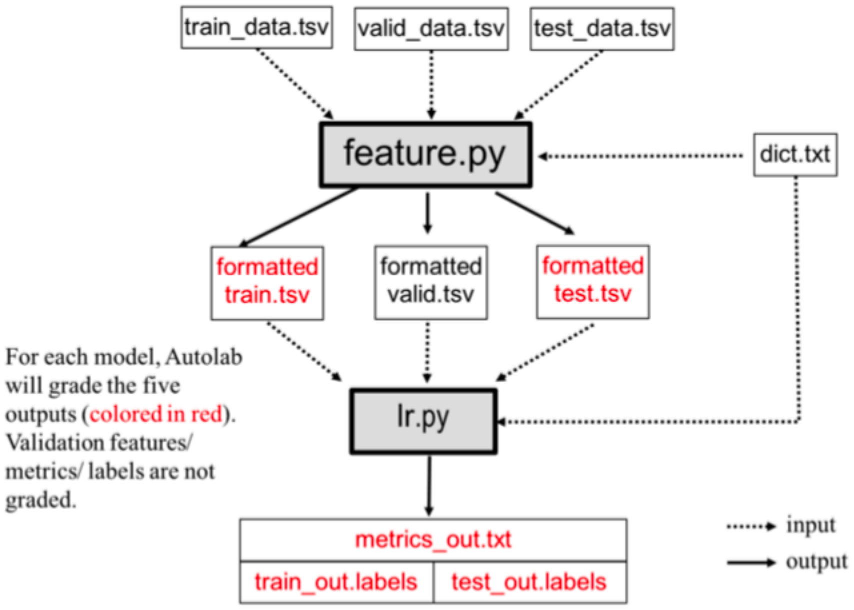

Figure 2.1: Programming pipeline for sentiment analyzer based on binary logistic regression

This first program is `feature.{py|java|cpp|m}`, that converts raw data (e.g., `train_data.tsv`, `valid_data.tsv`, and `test_data.tsv`) into formatted training, validation and test data based on the vocabulary information in the dictionary file `dict.txt`. To be specific, this program is to transfer the whole movie review text into a feature vector using some feature extraction methods. The formatted datasets should be stored in .tsv format. Details of formatted datasets will be introduced in Section 2.4.2 and Section 2.5.1.

The second program is `lr.{py|java|cpp|m}`, that implements a sentiment polarity analyzer using binary logistic regression. The file should learn the parameters of a binary logistic regression model that predicts a sentiment polarity (i.e. label) for the corresponding feature vector of each movie review. The program should output the labels of the training and test examples and calculate training and test error (percentage of incorrectly labeled reviews). As discussed in Appendix A.2 and A.3, efficient computation can be obtained with the help of the indexing information in the dictionary file `dict.txt`.

### 2.4.2 Feature Engineering

Your implementation of `feature.{py|java|cpp|m}` should have an input argument `<feature_flag>` that specifies one of two types of feature extraction structures that should be used by the logistic regression model. The two structures are illustrated below as probabilities of the labels given the inputs.

**Model 1** $p(y^{(i)} \mid \mathbf{1}_{\mathbf{occur}}(\mathbf{x^{(i)}}, \mathbf{Vocab}), \boldsymbol{\theta})$: This model defines a probability distribution over the current label $y^{(i)}$ using the parameters $\boldsymbol{\theta}$ and a *bag-of-word* feature vector $\mathbf{1_{occur}}(\mathbf{x^{(i)}}, \mathbf{Vocab})$ indicating which word in vocabulary $\mathbf{Vocab}$ of the dictionary occurs at least once in the movie review example $\mathbf{x}^{(i)}$. The entry in the indicator vector associated to the occurring word will set to one (otherwise, it is zero). This bag-of-word model should be used when `<feature_flag>` is set to 1.

**Model 2** $p(y^{(i)} \mid \mathbf{1_{trim}}(\mathbf{x^{(i)}}, \mathbf{Vocab}, t), \boldsymbol{\theta})$: This model defines a probability distribution over the current label $y^{(i)}$ using the parameters $\boldsymbol{\theta}$ and a *trimmed* bag-of-word feature vector $\mathbf{1_{trim}}(\mathbf{x^{(i)}}, \mathbf{Vocab}, t)$ indicating (1) which word in vocabulary $\mathbf{Vocab}$ of the dictionary occurs in the movie review example $\mathbf{x}^{(i)}$, AND (2) the *count of the word* is LESS THAN ($<$) threshold $t$. The entry in the indicator vector associated to the word that satisfies both conditions will set to one (otherwise, it is zero, including no shown and high-frequent words). This trimmed bag-of-word model should be used when `<feature_flag>` is set to 2. In this assignment, use the constant trimming threshold $t = 4$.

The motivation of Model 2 is that keywords that truly represent the sentiment may not occur too frequently, this trimming strategy can make the feature presentation cleaner by removing highly repetitive words that are useless and neutral, such ”the”, ”a”, ”to”, etc. You will observe whether this basic and heuristic strategy based on this intuition will bring in performance improvement.

Note that above $\mathbf{1_{occur}}$ and $\mathbf{1_{trim}}$ are described as a dense feature representation as showed in Tables A.2 for illustration purpose. In your implementation, you should further convert it to the representation in A.3 for Model 1 and the representation in A.5 for Model 2, such that the formatted data outputs match Section 2.5.1.

### 2.4.3 Command Line Arguments

The autograder runs and evaluates the output from the files generated, using the following command (note `feature` will be run before `lr`):

```
For Python: $ python feature.py [args1...]
            $ python lr.py [args2...]
For Java:   $ java feature.java [args1...]
            $ java lr.java [args2...]
For C++:    $ g++ feature.cpp ./a.out [args1...]
            $ g++ lr.cpp ./a.out [args2...]
For Octave: $ octave -qH feature.m [args1...]
            $ octave -qH lr.m [args2...]
```

Where above `[args1...]` is a placeholder for eight command-line arguments:`<train_input>` `<validation_input>` `<test_input>` `<dict_input>` `<formatted_train_out>` `<formatted_validation_out>` `<formatted_test_out>` `<feature_flag>`. These arguments are described in detail below:

1. `<train_input>`: path to the training input `.tsv` file (see Section 2.2)
2. `<validation_input>`: path to the validation input `.tsv` file (see Section 2.2)
3. `<test_input>`: path to the test input `.tsv` file (see Section 2.2)
4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2.2)
5. `<formatted_train_out>`: path to output `.tsv` file to which the feature extractions on the *training* data should be written (see Section 2.5.1)

6. `<formatted_validation_out>`: path to output `.tsv` file to which the feature extractions on the *validation* data should be written (see Section 2.5.1)

7. `<formatted_test_out>`: path to output `.tsv` file to which the feature extractions on the *test* data should be written (see Section 2.5.1)

8. `<feature_flag>`: integer taking value 1 or 2 that specifies whether to construct the Model 1 feature set or the Model 2 feature set (see Section 2.4.2)—that is, if `feature_flag`==1 use Model 1 features; if `feature_flag`==2 use Model 2 features

On the other hand, `[args2...]` is a placeholder for eight command-line arguments:`<formatted_train_input>` `<formatted_validation_input>` `<formatted_test_input>` `<dict_input>` `<train_out>` `<test_out>` `<metrics_out>` `<num_epoch>`. These arguments are described in detail below:

1. `<formatted_train_input>`: path to the formatted training input `.tsv` file (see Section 2.5.1)

2. `<formatted_validation_input>`: path to the formatted validation input `.tsv` file (see Section 2.5.1)

3. `<formatted_test_input>`: path to the formatted test input `.tsv` file (see Section 2.5.1)

4. `<dict_input>`: path to the dictionary input `.txt` file (see Section 2.2)

5. `<train_out>`: path to output `.labels` file to which the prediction on the *training* data should be written (see Section 2.5.2)

6. `<test_out>`: path to output `.labels` file to which the prediction on the *test* data should be written (see Section 2.5.2)

7. `<metrics_out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 2.5.3)

8. `<num_epoch>`: integer specifying the number of times SGD loops through all of the training data (e.g., if `<num_epoch>` equals 5, then each training example will be used in SGD 5 times).

As an example, if you implemented your program in Python, the following two command lines would run your programs on the data provided in the handout for 60 epochs using the features from Model 1.

```
$ python feature.py train_data.tsv valid_data.tsv test_data.tsv \
dict.txt formatted_train.tsv formatted_valid.tsv formatted_test.tsv 1

$ python lr.py formatted_train.tsv formatted_valid.tsv formatted_test\
.tsv dict.txt train_out.labels test_out.labels metrics_out.txt 60
```

## 2.5 Program Outputs

### 2.5.1 Output: Formatted Data Files

Your `feature` program should write three output `.tsv` files converting original data to formatted data on `<formatted_train_out>`, `<formatted_valid_out>`, and `<formatted_test_out>`. Each should contain the formatted presentation for each example printed on a new line. Use `\n` to create a new line. The format for each line should exactly match

label`\t`index[word1]:value1`\t`index[word2]:value2`\t`...index[wordM]:valueM`\n`

Where above, the first column is label, and the rest are "index[word]:value" feature elements. index[word] is the index of the word in the dictionary, and value is the value of this feature (in this assignment, the value is one or zero). There is a colon, `:`, between index[word] and corresponding value. Columns are separated using a table character, `\t`. The handout contains example `<formatted_train_out>`, `<formatted_valid_out>`, and `<formatted_test_out>` for your reference.

The formatted output will be checked separately by the autograder by running your `feature` program on some unseen datasets and evaluating your output file against the reference formatted files. Examples of content of formatted output file are given below.

```
0       2915:1  21514:1 166:1   32:1    10699:1 305:1   ...
0       7723:1  51:1    8701:1  74:1    370:1   8:1    ...
1       229:1   48:1    326:1   43:1    576:1   55:1    ...
1       8126:1  1349:1  58:1    4709:1  48:1    8319:1  ...
```

### 2.5.2 Output: Labels Files

Your `lr` program should produce two output `.labels` files containing the predictions of your model on training data (`<train_out>`) and test data (`<test_out>`). Each should contain the predicted labels for each example printed on a new line. Use `\n` to create a new line.

Your labels should exactly match those of a reference implementation – this will be checked by the autograder by running your program and evaluating your output file against the reference solution. Examples of the content of the output file are given below.

```
0
0
1
0
```

### 2.5.3 Output Metrics

Generate a file where you report the following metrics:

**error** After the final epoch (i.e. when training has completed fully), report the final training error `error(train)` and test error `error(test)`.

All of your reported numbers should be within 0.01 of the reference solution. The following is the reference solution for large dataset with Model 1 feature structure after 60 training epochs. See `model1_metrics_out.txt` in the handout.

```
error(train): 0.074167
error(test): 0.247500
```

Take care that your output has the exact same format as shown above. Each line should be terminated by a Unix line ending `\n`. There is a whitespace character after the colon.

## A Implementation Details for Logistic Regression

### A.1 Examples of Features

Here we provide examples of the features constructed by Model 1 and Model 2. Table A.1 shows an example input file, where column $i$ indexes the $i$th movie review example. Rather than working directly with this input file, you should transform from the sentiment/text representation into a label/feature vector representation.

Table A.2 shows the dense occurrence-indicator representation expected for Model 1. The size of each feature vector (i.e. number of feature columns in the table) is equal to the size of the entire vocabulary of words stored in the given `dict.txt` (this dictionary is actually constructed from the same training data in `largeset`). Each row corresponds to a single example, which we have indexed by $i$.

It would be *highly impractical* to actually store your feature vectors $\mathbf{x}^{(i)} \in \mathbb{R}^M$ in the dense representation shown in Table A.2 which takes $O(M)$ space per vector ($M$ is around 40 thousands for the dictionary). This is because the features are extremely sparse: for the second example ($i = 2$), only three of the features is non-zero for Model 1 and only two for Model 2. As such, we now consider a sparse representation of the features that will save both memory and computation.

Table A.3 shows the sparse representation (bag-of-word representation) of the feature vectors. Each feature vector is now represented by a map from the index of the feature (e.g. index[`"apple"`]) to its value which is 1. The space savings comes from the fact that we can omit from the map any feature whose value is zero. In this way, the map only contains *non-zero entry* for each Model 1 feature vector.

Using the same sparse representation of features, we present an example of the features used by Model 2. This involves two step: (1) construct the count-of-word representation of the feature vector (see Table A.4); (2) trim/remove the highly repetitive words/features and set the value of all remaining features to one (see Table A.5).

### A.2 Efficient Computation of the Dot-Product

In simple linear models like logistic regression, the computation is often dominated by the dot-product $\boldsymbol{\theta}^T\mathbf{x}$ of the parameters $\boldsymbol{\theta} \in \mathbb{R}^M$ with the feature vector $\mathbf{x} \in \mathbb{R}^M$ . When a dense representation of $\mathbf{x}$ (such as that shown in Table A.2) is used, this dot-product requires $O(M)$ computation. Why? Because the dot-product requires a sum over each entry in the vector:

$$\boldsymbol{\theta}^T\mathbf{x} = \sum_{m=1}^{M} \theta_m x_m \tag{A.1}$$

However, if our feature vector is represented sparsely, we can observe that the only elements of the feature vector that will contribute a non-zero value to the sum are those where $x_m \neq 0$, since this would allow $\theta_m x_m$ to be nonzero. As such, we can write the dot-product as below:

$$\boldsymbol{\theta}^T\mathbf{x} = \sum_{m\in\{1,\ldots,M\} \text{ s.t. } x_m\neq 0} \theta_m x_m \tag{A.2}$$

This requires only computation proportional to the number of non-zero entries in $\mathbf{x}$, which is generally very small for Model 1 and Model 2 compared to the size of the vocabulary. To ensure that your code runs quickly it is best to write the dot-product in the latter form (Equation (A.2)).

## A.3 Data Structures for Fast Dot-Product

Lastly, there is a question of how to implement this dot-product efficiently in practice. The key is choosing appropriate data structures. The most common approach is to choose a dense representation for $\boldsymbol{\theta}$. In C++ or Java, you could choose an array of `float` or `double`. In Python, you could choose a `numpy` array or a list.

To represent your feature vectors, you might need multiple data structures. First, you could create a shared mapping from a feature name (e.g. `apple` or `boy`) to the corresponding index in the dense parameter vector. This shared mapping has already been provided to you in the `dict.txt`, and you can extract the index of the word from the dictionary file for all later computation. In fact, you should be able to construct the dictionary on your own from the training data (we have done this step for you in the handout). Once you know the size of this mapping (which is the size of the dictionary file), you know the size of the parameter vector $\boldsymbol{\theta}$.

Another data structure should be used to represent the feature vectors themselves. This assignment use the option to directly store a mapping from the integer index in the dictionary mapping (i.e. the index $m$) to the value of the feature $x_m$. Only the indexs of words satisfying certain conditions will be stored, and all other indexs are implies to have zero value of the feature $x_m$. This structure option will ensure that your code runs fast so long as you are doing an efficient computation instead of the $O(M)$ version.

**Note for out-of-vocabulary features** The dictionary in the handout is made from the same training data in the large data set. You may encounter some words in the validation data and the test data that do not appear in the vocabulary mapping. In this assignment, you should ignore those words during prediction and evaluation.

| **example index** $i$ | **sentiment** $y^{(i)}$ | **review text** $\mathbf{x}^{(i)}$ |
|---|---|---|
| 1 | pos | apple boy , cat dog |
| 2 | pos | boy boy : dog dog ; dog dog . dog egg egg |
| 3 | neg | apple apple apple apple boy cat cat dog |
| 4 | neg | egg fish |

Table A.1: Abstract representation of the input file format. The $i$th row of this file will be used to construct the $i$th training example using either Model 1 features (Table A.3) or Model 2 features (Table A.5).

| $i$ | **label** $y^{(i)}$ | **features** $\mathbf{x}^{(i)}$ | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | `zoo` | ... | `apple` | `boy` | `cat` | `dog` | `egg` | `fish` | `girl` | `head` | ... | `zero` |
| 1 | 1 | 0 | ... | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | ... | 0 |
| 2 | 1 | 0 | ... | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | ... | 0 |
| 3 | 0 | 0 | ... | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | ... | 0 |
| 4 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | ... | 0 |

Table A.2: Dense feature representation for Model 1 corresponding to the input file in Table A.1. The $i$th row corresponds to the $i$th training example. Each dense feature has the size of the vocabulary in the dictionary. Punctuations are excluded.

| $i$ | **label** $y^{(i)}$ | **features** $\mathbf{x}^{(i)}$ |
|---|---|---|
| 1 | 1 | { index[“`apple`”]: 1, index[“`boy`”]: 1, index[“`cat`”]: 1, index[“`dog`”]: 1 } |
| 2 | 1 | { index[“`boy`”]: 1, index[“`dog`”]: 1, index[“`egg`”]: 1 } |
| 3 | 0 | { index[“`apple`”]: 1, index[“`boy`”]: 1, index[“`cat`”]: 1, index[“`dog`”]:1 } |
| 4 | 0 | { index[“`egg`”]: 1, index[“`fish`”]: 1 } |

Table A.3: Sparse feature representation (bag-of-word representation) for Model 1 corresponding to the input file in Table A.1.

| $i$ | **label** $y^{(i)}$ | **features** $\mathbf{x}^{(i)}$ |
|---|---|---|
| 1 | 1 | { index[“`apple`”]: 1, index[“`boy`”]: 1, index[“`cat`”]: 1, index[“`dog`”]: 1 } |
| 2 | 1 | { index[“`boy`”]: 2, index[“`dog`”]: 5, index[“`egg`”]: 2 } |
| 3 | 0 | { index[“`apple`”]: 4, index[“`boy`”]: 1, index[“`cat`”]: 2, index[“`dog`”]: 1 } |
| 4 | 0 | { index[“`egg`”]: 1, index[“`fish`”]: 1 } |

Table A.4: Count of word representation for Model 2 corresponding to the input file in Table A.1.

| $i$ | **label** $y^{(i)}$ | **features** $\mathbf{x}^{(i)}$ |
|---|---|---|
| 1 | 1 | { index[“`apple`”]: 1, index[“`boy`”]: 1, index[“`cat`”]: 1, index[“`dog`”]: 1 } |
| 2 | 1 | { index[“`boy`”]: 1, index[“`egg`”]:1 } |
| 3 | 0 | { index[“`boy`”]: 1, index[“`cat`”]: 1, index[“`dog`”]: 1 } |
| 4 | 0 | { index[“`egg`”]: 1, index[“`fish`”]: 1 } |

Table A.5: Sparse feature representation for Model 2 corresponding to the input file in Table A.1. Assume that the trimming threshold is 4. As a result, ”dog” in example 2 and ”apple” in example 3 are removed and the value of all remaining features are reset to value 1.