DS-GA-1011: Natural Language Processing with Representation Learning, Fall 2024

Representing and Classifying Text

Aditeya Baral
N19186654

Please write down any collaborators, AI tools (ChatGPT, Copliot, codex, etc.), and external resources you used for this assignment here.
Collaborators:
AI tools:
Resources:

By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

Welcome to your first assignment! The goal of this assignment is to get familiar with basic text classification models and explore word vectors using basic linear algebra tools. **Before you get started, please read the Submission section thoroughly**.

Due Date - 11:59 PM 09/20/2024

Submission

Submission is done on Gradescope.

Written: When submitting the written parts, make sure to select **all** the pages that contain part of your answer for that problem, or else you will not get credit. You can either directly type your solution between the `shaded` environments in the released `.tex` file, or write your solution using a pen or stylus. A `.pdf` file must be submitted.
Programming: Questions marked with "coding" next to the assigned to the points require a coding part in `submission.py`. Submit `submission.py` and we will run an autograder on Gradescope. You can use functions in `util.py`. However, please do not import additional libraries (e.g. `sklearn`) that aren't mentioned in the assignment, otherwise the grader may crash and no credit will be given. You can run `test_classification.py` and `test_embedding.py` to test your code but you don't need to submit it.

Problem 1: Naive Bayes classifier

In this problem, we will study the *decision boundary* of multinomial Naive Bayes model for binary text classification. The decision boundary is often specificed as the level set of a function: $\{x \in \mathcal{X} : h(x) = 0\}$, where x for which $h(x) > 0$ is in the positive class and x for which $h(x) < 0$ is in the negative class.

1. [2 points] Give an expression of $h(x)$ for the Naive Bayes model $p_\theta(y \mid x)$, where θ denotes the parameters of the model.

From Bayes' Theorem, we know that the conditional probability of $p_\theta(y \mid x)$, where θ denotes the parameters of the model is:

$$p_\theta(y \mid x) = \frac{p_\theta(x \mid y) \cdot p_\theta(y)}{p_\theta(x)} \tag{1}$$

where $p_\theta(x \mid y)$ is the probability of observing the sequence x given a class y. If we consider the **Naive Bayes' assumption** to hold, i.e **the probability of observing each token is independent**, this can be estimated to be the conditional probability of observing each token x_i in the sequence x such that:

$$p_\theta(x \mid y) = \prod_{i=1}^{n} p_\theta(x_i \mid y) \tag{2}$$

$p_\theta(x)$ is the probability of observing the sequence x and can be expressed as the **sum of the joint probabilities of observing x over all classes y.**

$$p_\theta(x) = \sum_{y} p_\theta(x \mid y) \cdot p_\theta(y) \tag{3}$$

However, since Eq. 3 is a **constant value** for any given x and we are only concerned with comparing the probabilities of a sequence x belonging to a particular class, it can be ignored. This simplifies Eq. 1 to:

$$p_\theta(y \mid x) \approx p_\theta(y) \prod_{i=1}^{n} p_\theta(x_i \mid y) \tag{4}$$

At the decision boundary, $h(x) = 0$. This implies that **the probability of sequence x is equal in both classes** such that:

$$p_\theta(y = 0 \mid x) = p_\theta(y = 1 \mid x) \tag{5}$$

On substituting Eq. 4 in the equation above, we obtain the following:

$$p_\theta(y = 0) \prod_{i=1}^{n} p_\theta(x_i \mid y = 0) = p_\theta(y = 1) \prod_{i=1}^{n} p_\theta(x_i \mid y = 1) \tag{6}$$

Applying log on both sides, this can be simplified to:

$$\log p_\theta(y = 0) + \sum_{i=1}^{n} \log p_\theta(x_i \mid y = 0) = \log p_\theta(y = 1) + \sum_{i=1}^{n} \log p_\theta(x_i \mid y = 1) \tag{7}$$

Thus, the **expression $h(x)$ for the Naive Bayes model $p_\theta(y \mid x)$, where θ denotes the parameters of the model can be defined below**

$$h(x) = \sum_{i=1}^{n} \log p_\theta(x_i \mid y = 1) - \sum_{i=1}^{n} \log p_\theta(x_i \mid y = 0) + \log p_\theta(y = 1) - \log p_\theta(y = 0) \tag{8}$$

If we assume $p_\theta(y = 1) = \log p_\theta(y = 0)$, the above equation can be further simplified to:

$$h(x) = \sum_{i=1}^{n} \log p_\theta(x_i \mid y = 1) - \sum_{i=1}^{n} \log p_\theta(x_i \mid y = 0) \tag{9}$$

2. [3 points] Recall that for multinomial Naive Bayes, we have the input $X = (X_1, \ldots, X_n)$ where n is the number of words in an example. In general, n changes with each example but we can ignore that for now. We assume that $X_i \mid Y = y \sim \text{Categorical}(\theta_{w_1,y}, \ldots \theta_{w_m,y})$ where $Y \in \{0, 1\}$, $w_i \in \mathcal{V}$, and $m = |\mathcal{V}|$ is the vocabulary size. Further, $Y \sim \text{Bernoulli}(\theta_1)$. Show that the multinomial Naive Bayes model has a linear decision boundary, i.e. show that $h(x)$ can be written in the form $w \cdot x + b = 0$. [**RECALL:** The categorical distribution is a multinomial distribution with one trial. Its PMF is

$$p(x_1, \ldots, x_m) = \prod_{i=1}^{m} \theta_i^{x_i} \ ,$$

where $x_i = \mathbb{1}[x = i]$, $\sum_{i=1}^{m} x_i = 1$, and $\sum_{i=1}^{m} \theta_i = 1$.]

From Bayes' Theorem (Eq. 1), we know that the conditional probability of $P(Y \mid X)$ is given by:

$$P(Y \mid X) = \frac{P(X \mid Y) \cdot P(Y)}{P(X)} \tag{10}$$

Given $X = (X_1, \ldots, X_n)$ and $p(x_1, \ldots, x_m) = \prod_{i=1}^{m} \theta_i^{x_i}$, for any given class y, we can represent $P(X \mid Y = y)$ to be:

$$P(X \mid Y = y) = \prod_{i=1}^{n} \theta_{X_i = w_i, y} \tag{11}$$

Since $Y \sim \text{Bernoulli}(\theta_1)$, we can also derive the following probabilities:

$$P(Y = 1) = \theta_1 \tag{12}$$

$$P(Y = 0) = 1 - \theta_1 \tag{13}$$

Substituting these in Eq. 10, we can simplify the expressions $P(Y = 0 \mid X)$ and $P(Y = 1 \mid X)$ to be:

$$P(Y = 0 \mid X) = \frac{(1 - \theta_1) \cdot \prod_{i=1}^{n} \theta_{w_i,0}}{P(X)} \tag{14}$$

and

$$P(Y = 1 \mid X) = \frac{\theta_1 \cdot \prod_{i=1}^{n} \theta_{w_i,1}}{P(X)} \tag{15}$$

Using the Naive Bayes model, at the decision boundary $h(x) = 0$, we assign equal probabilities for each class, i.e $P(Y = 1 \mid X) = P(Y = 0 \mid X)$. This can be expressed as:

$$\theta_1 \cdot \prod_{i=1}^{n} \theta_{w_i,1} = (1 - \theta_1) \cdot \prod_{i=1}^{n} \theta_{w_i,0} \tag{16}$$

We can apply log on both sides and simplify the expression as:

$$\log \theta_1 + \sum_{i=1}^{n} \log \theta_{w_i,1} = \log(1 - \theta_1) + \sum_{i=1}^{n} \log \theta_{w_i,0} \tag{17}$$

Rearranging and grouping the terms such that we obtain the following:

$$\sum_{i=1}^{n} (\log \theta_{w_i,1} - \log \theta_{w_i,0}) + \log \theta_1 - \log(1 - \theta_1) = 0 \tag{18}$$

Applying logarithmic properties, this can be expressed as:

$$\sum_{i=1}^{n} \log \frac{\theta_{w_i,1}}{\theta_{w_i,0}} + \log \frac{\theta_1}{1 - \theta_1} = 0 \tag{19}$$

Let:

$$w_i = \log \frac{\theta_{w_i,1}}{\theta_{w_i,0}} \tag{20}$$

$$b = log \frac{\theta_1}{1-\theta_1} \tag{21}$$

The expression can now be represented as the following where w_i is the vector of weights, N_j represents the input features (here, it represents the counts of each word j in the sentence) and b is the bias.

$$\sum_{i=j}^{m} w_j \cdot N_j + b = 0 \tag{22}$$

We can thus express the decision boundary $h(x)$ by the following expression, where x in the input feature vector which comprises the counts of each word in the sentence. **Since we have represented the decision function in the form of $w \cdot x + b = 0$, this proves that the multinomial Naive Bayes model has a linear decision boundary**.

$$\sum_{j=1}^{m} w_j \cdot x + b = 0 \tag{23}$$

3. [2 points] In the above model, X_i represents a single word, i.e. it's a unigram model. Think of an example in text classification where the Naive Bayes assumption might be violated. How would you allieviate the problem?

(a) The Naive Bayes model assumes **all input features are conditionally independent given a label** y. This implies that all words in a sequence are conditionally independent in an unigram model. Although this works in some cases, it fails in most real-world instances when **words are related to each other** or there is a **contextual flow** of information between words, since semantics often depends on patterns or context in a language.

For example, consider the following two sentences:

i. "I am not happy with the service".

ii. "I am happy with the service".

In the first sentence, the word "not" **negates** the word "happy", which changes the sentiment of the sentence. However, in the unigram model, the word "not" is treated as an independent feature and does not affect the occurrence of the word "happy", while their **co-occurrence and dependency should be factored** into the model's predictions.

(b) This assumption also fails for **synonyms or words with similar meanings**. For example, consider the following two pairs of sentences:

i. "I love dogs! They are adorable."

ii. "I adore canines! They are cute."

i. "Sam is a funny comedian. I find him hilarious."

ii. "Sam's shows are really fun"

In both cases, the words "love" and "adore" have similar meanings, as do "dogs" and "canines". Similarly, "funny" and "fun" are related, as are "hilarious" and "fun". However, in the unigram model, these words are treated as **independent features** and do not capture the semantic similarity between them.

(c) To address these limitations, we can use a **bigram (or higher n-gram) model** that considers word pairs as features and thus captures short-range dependencies. This allows the model to **preserve the contextual flow of information between words and their relationships**. For example, in the sentence "I am not happy with the service", the bigram model can capture the relationship between the words "not" and "happy" and correctly predict the sentiment of the sentence. To address synonyms or words with similar meanings, we can employ better feature engineering techniques to learn richer representations, (for example, using **word embeddings**) to **capture the semantic similarity between words** (although this will *not* completely remove the independence assumption) and improve the model's performance.

4. [2 points] Since the decision boundary is linear, the Naive Bayes model works well if the data is linearly separable. Discuss ways to make text data works in this setting, i.e. make the data more linearly separable and its influence on model generalization.

(a) **Learning a reduced feature space**: Learning features for every word in the vocabulary results in **high dimensional data** as well as the **lack of contextually distinguishing information** in the learnt features. By opting to convert all words to lowercase, removing stopwords and performing basic **text preprocessing** such as stemming or lemmatising, we can reduce noise, decrease the size of the feature space and reduce their sparsity, thus making the decision boundary simpler to compute and more linear. These steps also help **decrease the vocabulary size** and aid the model to generalise better by learning features only for the root forms of the words, thus preventing the conditional independence assumption from strongly affecting the model when synonyms or root-form derivatives are frequently used. We can also try reducing the dimensions of the feature space using **dimensionality reduction** techniques such as **SVD**.

(b) **Learning more impactful features**: Using raw counts of words can be problematic because the model cannot distinguish between important and unimportant features, especially ones which can help distinguish between classes. Raw counts also lead to common words (which are non-distinguishing in nature) to possess larger weight values and thus cause them to play a larger role in classification. Scaling the features using techniques such as the **Term Frequency-Inverse Document Frequency (TF-IDF)** approach can **emphasise and assign larger weights to more important or impactful words** (often the rare words) in a sentence, thus making the decision boundary more linear between two classes and thus making it easier to model it using Naive Bayes. We can also use **pre-trained word embeddings to account for semantic similarities** between words (such as synonyms) and thus help increase the separation between opposite classes.

(c) **Better feature engineering**: Instead of a unigram model, we can choose to learn **bigrams (or higher level n-grams)** for our feature space. Although this increases the dimension of our feature space, by experimenting with the value of n, we can optimise and choose such a value that allows the model to **capture patterns, short-range dependencies and informative contextual information** thus improving performance on real-world tasks without the risk of over-fitting on specific patterns or a significant increase in computational complexity.

Problem 2: Natural language inference

In this problem, you will build a logistic regression model for textual entailment. Given a *premise* sentence, and a *hypothesis* sentence, we would like to predict whether the hypothesis is *entailed* by the premise, i.e. if the premise is true, then the hypothesis must be true.
Example:

label	premise	hypothesis
entailment	The kids are playing in the park	The kids are playing
non-entailment	The kids are playing in the park	The kids are happy

1. [1 point] Given a dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ where $y \in \{0, 1\}$, let ϕ be the feature extractor and w be the weight vector. Write the maximum log-likelihood objective as a *minimization* problem.

The **maximum log-likelihood** objective $MLE(w; D)$ of a logistic regression model for a binary classification problem can be expressed as the following where σ is the sigmoid function, ϕ is the feature extractor and w is the weight vector.

$$MLE(w; D) = \sum_{i=1}^{n} \left[y^{(i)} \log \sigma(w^T \phi(x^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(w^T \phi(x^{(i)}))) \right] \tag{24}$$

We can thus convert the **MLE into a *minimization* problem by obtaining the negative of the log-likelihood** $l(w) = -MLE(w; D)$, expressed by the following:

$$l(w) = -MLE(w; D) = \sum_{i=1}^{n} \left[-y^{(i)} \log \sigma(w^T \phi(x^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(w^T \phi(x^{(i)}))) \right] \tag{25}$$

2. [3 point, coding] We first need to decide the features to represent x. Implement `extract_unigram_features` which returns a BoW feature vector for the premise and the hypothesis.

```python
def extract_unigram_features(ex):
    """Return unigrams in the hypothesis and the premise.
    Parameters:
        ex : dict
            Keys are gold_label (int, optional), sentence1 (list), and sentence2 (list)
    Returns:
        A dictionary of BoW featurs of x.
    Example:
        "I love it", "I hate it" --> {"I":2, "it":2, "hate":1, "love":1}
    """
    # a dictionary of BoW features for each token in the hypothesis and the premise
    unigram_features = dict()
    premise = ex["sentence1"]
    hypothesis = ex["sentence2"]
    for token in premise + hypothesis:
        # increment the count of the token in the dictionary
        unigram_features[token] = unigram_features.get(token, 0) + 1
    return unigram_features
```

3. [2 point] Let $\ell(w)$ be the objective you obtained above. Compute the gradient of $\ell_i(w)$ given a single example $(x^{(i)}, y^{(i)})$. Note that $\ell(w) = \sum_{i=1}^{n} \ell_i(w)$. You can use $f_w(x) = \frac{1}{1+e^{-w\cdot\phi(x)}}$ to simplify the expression.

From Eq. 25, we know that $l_i(w)$ for a single example can be expressed as:

$$l_i(w) = -\left[y^{(i)} \log \sigma(w^T \phi(x^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(w^T \phi(x^{(i)})))\right] \tag{26}$$

We can substitute the sigmoid function defined by $\sigma(w^T \phi(x^{(i)})) = \frac{1}{1+e^{-w\cdot\phi(x^{(i)})}} = f_w(x^{(i)})$ in the expression to simplify it further:

$$l_i(w) = -y^{(i)} \log f_w(x^{(i)}) - (1 - y^{(i)}) \log(1 - f_w(x^{(i)})) \tag{27}$$

$$\nabla l_i(w) = \frac{\partial l_i(w)}{\partial w} = -y^{(i)} \frac{\partial}{\partial w} \log f_w(x^{(i)}) - (1 - y^{(i)}) \frac{\partial}{\partial w} \log(1 - f_w(x^{(i)})) \tag{28}$$

$$\nabla l_i(w) = -y^{(i)} \frac{1}{f_w(x^{(i)})} \frac{\partial f_w(x^{(i)})}{\partial w} - (1 - y^{(i)}) \frac{1}{1 - f_w(x^{(i)})} \frac{\partial (1 - f_w(x^{(i)}))}{\partial w} \tag{29}$$

$$\nabla l_i(w) = -y^{(i)} \frac{1}{f_w(x^{(i)})} \frac{\partial f_w(x^{(i)})}{\partial z} \frac{\partial z}{\partial w} + (1 - y^{(i)}) \frac{1}{1 - f_w(x^{(i)})} \frac{\partial f_w(x^{(i)})}{\partial z} \frac{\partial z}{\partial w} \tag{30}$$

where $z = w \cdot \phi(x)$. Thus, we can also compute $\frac{\partial f_w(x^{(i)})}{\partial z}$ and $\frac{\partial z}{\partial w}$ to be:

$$\frac{\partial f_w(x^{(i)})}{\partial z} = f_w(x^{(i)})(1 - f_w(x^{(i)})) \tag{31}$$

$$\frac{\partial z}{\partial w} = \phi(x^{(i)}) \tag{32}$$

Substituting these back into the above expression $\nabla l_i(w)$, we obtain the following expression:

$$\nabla l_i(w) = -y^{(i)} \frac{f_w(x^{(i)})(1 - f_w(x^{(i)})) \cdot \phi(x^{(i)})}{f_w(x^{(i)})} + (1 - y^{(i)}) \frac{f_w(x^{(i)})(1 - f_w(x^{(i)})) \cdot \phi(x^{(i)})}{1 - f_w(x^{(i)})} \tag{33}$$

We can simplify it using the following steps:

$$\nabla l_i(w) = \left[-\frac{y^{(i)}}{f_w(x^{(i)})} + \frac{1 - y^{(i)}}{1 - f_w(x^{(i)})}\right] f_w(x^{(i)})(1 - f_w(x^{(i)})) \cdot \phi(x^{(i)}) \tag{34}$$

Thus, **the gradient $\nabla l_i(w)$ can be expressed as**:

$$\nabla l_i(w) = (f_w(x^{(i)}) - y^{(i)}) \cdot \phi(x^{(i)}) \tag{35}$$

4. [5 points, coding] Use the gradient you derived above to implement `learn_predictor`. You must obtain an error rate less than 0.3 on the training set and less than 0.4 on the test set to get full credit *using the unigram feature extractor*.

```python
def learn_predictor(
    train_data, valid_data, feature_extractor, learning_rate, num_epochs
):
    """Running SGD on training examples using the logistic loss.
    You may want to evaluate the error on training and dev example after each epoch.
    Take a look at the functions predict and evaluate_predictor in util.py,
    which will be useful for your implementation.
    Parameters:
        train_data : [{gold_label: {0,1}, sentence1: [str], sentence2: [str]}]
        valid_data : same as train_data
        feature_extractor : function
            data (dict) --> feature vector (dict)
        learning_rate : float
        num_epochs : int
    Returns:
        weights : dict
            feature name (str) : weight (float)
    """
    # extract features from the training and validation data
    train_features = list(map(feature_extractor, train_data))
    valid_features = list(map(feature_extractor, valid_data))

    # create a vocabulary of all the features
    vocabulary = set()
    for feature in train_features:
        vocabulary.update(feature.keys())

    # initialize the weights of the features. We opt for zero initialization here.
    weights = {word: 0.0 for word in vocabulary}
    total_train_examples = len(train_data)
    total_valid_examples = len(valid_data)

    for epoch in range(num_epochs):
        # initialize the loss for the training and validation data
        training_loss = 0.0
        validation_loss = 0.0
        for i, ex in enumerate(train_data):
            label = ex["gold_label"]
            # predict the label using the weights
            prediction = predict(weights, train_features[i])
            # compute the loss and the gradient
            loss = -(label * np.log(prediction) + (1 - label) * np.log(1 - prediction))
            gradient = (prediction - label) * np.array(list(train_features[i].values()))
            for idx, word in enumerate(train_features[i]):
                # update the weights using the gradient (SGD)
                weights[word] -= learning_rate * gradient[idx]
            training_loss += loss
        # compute the average training loss
        training_loss /= total_train_examples
```

```
50
51          for i, ex in enumerate(valid_data):
52              label = ex["gold_label"]
53              # predict the label using the weights
54              prediction = predict(weights, valid_features[i])
55              # compute the loss
56              validation_loss += -(
57                  label * np.log(prediction) + (1 - label) * np.log(1 - prediction)
58              )
59          # compute the average validation loss
60          validation_loss /= total_valid_examples
61
62          # print the training and validation loss for each epoch
63          print(
64              f"Epoch {epoch}: training loss = {training_loss}, "
65              "validation loss = {validation_loss}"
66          )
67
68      return weights
```

5. [3 points] Discuss what are the potential problems with the unigram feature extractor and describe your design of a better feature extractor.

(a) **Does not capture context**: The unigram model **fails to capture patterns, phrases or short-range dependencies** to account for the contextual flow of information between words when they are related or collectively present an idea. For example, for the sentence: "I am not happy with the service.", the model will not be able to account for the **negation** of the word "happy", since each word is considered to be a unique feature. **An ideal feature extractor should be able to account for relationships between words**.

(b) **Does not prioritise impactful features**: The unigram feature selector being based on raw counts of words is **biased towards high-frequency words** comprising of stop words and uninformative words which do not help in classification. It also **treats each word equally** across sequences and does not account for the presence of a word across sequences. Words such as "the" and "it" are extremely frequent since they are used to form sentences and their corresponding word counts lead to higher magnitudes in a given sequence vector. These words are not useful for distinguishing sentences and may dilute the significance of more distinguishing words. **An ideal feature selector should be able to identify rare but meaningful words which help classify the sequence** easier and emphasise them.

(c) **Does not account for the relationship between sentences**: The unigram feature selector does not account for the relationship between sentences. **By factoring in information like the commonality of words, an ideal feature selector can easily find more contrast between the two** and thus distinguish between them.

We can address these problems by factoring in short-range dependencies and scaling the raw counts:

(a) **Capturing short-range dependencies using Bigrams**: This will help the model to **capture patterns and short-range dependencies** (such as negations) in a sentence. By treating a pair of words as a single feature and learning representations for these words, it can better **model the relationship between them** and leverage this for more accurate predictions.

(b) **Scaling the Raw Counts**: We can scale the raw term frequencies to make them more impactful by **incorporating information on sentence-level presence into the feature space**.

i. **Sentence Frequency**: This scale penalizes tokens that appear in both sentences, giving **higher weight to tokens that appear in only one sentence**. We add a `+0.001` in the denominator to prevent division by zero.

$$sf = \log\left(\frac{2}{\text{number of sentences containing the token} + 0.001}\right) \tag{36}$$

ii. **Token Frequency**: This scale helps emphasise rare tokens which might have a greater impact on distinguishing between both sentences and **reduces the weights of frequent words**. We add a `+0.001` in the denominator to prevent division by zero.

$$tf = \log\left(\frac{\text{total number of tokens}}{\text{number of times the token appears across both sentences} + 0.001}\right) \tag{37}$$

By scaling based on sentence frequency and token frequency, the **feature extractor emphasizes rare tokens which are either present in the *least* number of sentences and whenever are present, are present the *least* number of times** and gives them higher weights. The relation between the premise and hypothesis is also incorporated using the `sentence frequency` scale which helps **capture the contrast** between them.

6. [3 points, coding] Implement your feature extractor in `extract_custom_features`. You must get a lower error rate on the dev set than what you got using the unigram feature extractor to get full credits.

```python
def extract_custom_features(ex):
    """Returns features for a given pair of hypothesis and premise
    by performing the following:
    1. Extracting the unigrams and bigrams in the hypothesis and the premise.
    2. Scaling the features by the following two scales:
        - sentence frequency: log(number of sentences [= 2 in this case]
        / number of sentences containing the token)
        - token frequency: log(inverse of the fraction of the total number of tokens
        in the combined sentences that are the token)
    3. Multiplying the features by the two scales.

    Parameters:
        ex : dict
            Keys are gold_label (int, optional), sentence1 (list), and sentence2 (list)
    Returns:
        A dictionary of features for x.

    Example:
        "I love it", "I hate it" --> {
            "I": 2 * log(2/2.001) * log(10/2.001) = -0.001
            "it": 2 * log(2/2.001) * log(10/2.001) = -0.001
            "love": 1 * log(2/1.001) * log(10/1.001) = 1.5930
            "hate": 1 * log(2/1.001) * log(10/1.001) = 1.593
            ("I", "love"): 1 * log(2/1.001) * log(10/1.001) = 1.593
            ("love", "it"): 1 * log(2/1.001) * log(10/1.001) = 1.593
            ("I", "hate"): 1 * log(2/1.001) * log(10/1.001) = 1.593
            ("hate", "it"): 1 * log(2/1.001) * log(10/1.001) = 1.593
        }
    """
    custom_features = dict()

    premise = ex["sentence1"]
    hypothesis = ex["sentence2"]
    premise_bigrams = premise + ngrams(premise, 2)
    hypothesis_bigrams = hypothesis + ngrams(hypothesis, 2)
    combined = premise_bigrams + hypothesis_bigrams

    for token in combined:
        # increment the count of the token in the dictionary
        custom_features[token] = custom_features.get(token, 0) + 1

    total_tokens = len(combined)
    for token in custom_features:
        # we can scale the feature by multiplying the following two scales
        # intuition: we want the weight of a token to be maximum when it appears
        # in the fewest number of sentences and it appears as few times as possible
        # whenever it appears in a sentence. This is because the more unique a token,
        # the more it can help in distinguishing the sentences.
        sentence_frequency = np.log(
```

```
            2
            / (int(token in premise_bigrams) + int(token in hypothesis_bigrams) + 1e-3)
        )
        token_frequency = np.log(total_tokens / (custom_features[token] + 1e-3))
        custom_features[token] *= sentence_frequency * token_frequency
    return custom_features
```

7. [3 points] When you run the tests in `test_classification.py`, it will output a file `error_analysis.txt`. (You may want to take a look at the `error_analysis` function in `util.py`). Select five examples misclassified by the classifier using custom features. For each example, briefly state your intuition for why the classifier got it wrong, and what information about the example will be needed to get it right.

(a) **Premise**: A cabin shot of a very crowded airplane.
Hypothesis: The airplane is quite crowded.
Label: `1`
Prediction: `0`
Reason: From the dot product scores, we can see that thelowest scores comprise the impactful tokens such as "crowded", "very", "quite" and "airplane" (both ≈ 0). Since these do not have high dot product scores with the weights, their contribution to the overall predicted probability is also low and thus a lower probability score is computed, which implies that the model associates these terms with non-entailment. Additionally, the words "quite" and "very" have been used similarly and should have a high contribution as well as similarity score but their scores are low as well.
Solution: Learn richer feature representations through word embeddings which can assign higher weights to impactful words (like "crowded") that are relevant/common to both sentences and can draw a relationship between them, as well as account for synonyms and similar contexts such as "quite crowded" and "very crowded".

(b) **Premise**: A man is sitting outside next to fruit.
Hypothesis: A man takes a break next to his freshly picked fruit.
Label: `0`
Prediction: `1`
Reason: From the dot product scores, we can see that the highest scores comprise the words "sitting", "takes" and "man", which aren't sufficient to distinguish between the sentences, however the rare and impactful words like "break" and "fruit" do not have high dot product scores with the weights. Since the overall value of the dot product scores from the irrelevant words outweighs the overall value of the dot product scores from the important words due to the negative weight values for the important words, the overall predicted probability is high and entailment is predicted.
Solution: Learn richer feature representations through word embeddings which can assign higher weights to impactful words (like "break") that can contrast both sentences and can draw a relationship between them, as well as account for the semantic difference between phrases such as "take a break" and "sitting".

(c) **Premise**: A woman in a purple dress talks on her cellphone and a man reads a book as a two-story bus passes by outside their window.
Hypothesis: A woman calls her kids.
Label: `0`
Prediction: `1`
Reason: From the dot product scores, we can see that the highest scores comprise the words "her" and "woman", which aren't sufficient to distinguish between the sentences, however the rare and impactful words like "cellphone", "window", "kids" and "bus" do not have high dot product scores with the weights. Since the overall value of the dot product scores from the irrelevant words outweighs the overall value of the dot product scores from the important words, the overall predicted probability is also higher and entailment is predicted. Additionally, the model cannot draw a relationship between "talk on her cellphone" and "calls her kids", which might be related but semantically are different.
Solution: Learn richer feature representations which assign larger weights to impactful words (like "kids") through methods like word embeddings such that the distinguishing words can contribute more to the prediction and can contrast both sentences. Additionally, this would also handle spurious correlations such as the contextual difference between talking on the cellphone and calling someone.

(d) **Premise**: A taxi SUV drives past an urban construction site, as a man walks down the street in the other direction.
Hypothesis: An SUV and a man are going in opposite directions.
Label: `1`
Prediction: `0`
Reason: From the dot product scores, we can see that the highest scores comprise the words "urban" and "walks", which aren't sufficient to distinguish between the sentences, however the rare and impactful words like "man", "opposite" and "direction" do not have high dot product scores with the weights, which implies that the model associates these terms with non-entailment. Since the overall value of the dot product scores from the irrelevant words outweighs the overall value from the important words, the overall predicted probability is low and the prediction is incorrect.
Solution: Learn richer feature representations which assign larger weights to impactful words (like "man") and can associate these relationships with the second sentence. Better contextual understanding related to the words "going", "opposite" and "direction" are also needed, along with higher dot product scores for these words (which can be achieved through techniques like word embeddings) would also help the model draw a relationship between "going" and "walking" and associate them to the words "opposite" and "other" directions.

(e) **Premise**: A boy plays in the surf.
Hypothesis: The boy is wearing red trunks.
Label: `0`
Prediction: `1`
Reason: From the dot product scores, we can see that the highest scores comprise the words "boy", "wearing", "red" and "trunks", which aren't related to the premise but due to their high values, are associated with entailment. The word "boy" also has a very high feature value since it appears in both sentences and thus adds to the spurious correlation. Since the overall value of the dot product scores from the irrelevant words is high, the overall predicted probability is also high and thus the prediction is incorrect.
Solution: Learn richer feature representations through techniques like word embeddings which provide better contextual understanding contrasting the premise and hypothesis. Since words like "wearing" and "playing" aren't semantically similar concepts and there is no common contextual topic between the premise and hypothesis, the feature extractor should have accounted for mismatched information or the presence of irrelevant information and down-weighted these words significantly.

8. [3 points] Change `extract_unigram_features` such that it only extracts features from the hypothesis. How does it affect the accuracy? Is this what you expected? If yes, explain why. If not, give some hypothesis for why this is happening. Don't forget to change the function back before your submit. You do not need to submit your code for this part.

(a) The training loss decreases from `0.251` to `0.2468` and the validation loss decreases from `0.36` to `0.2805`.

(b) This is **unexpected** since we would expect the loss to increase because the **premise contains information required to predict whether the hypothesis entails it**. Without the premise, the model can only predict using the hypothesis and it is impossible to say whether it entails the premise without it. Since the relationship between the premise and hypothesis is required to contrast between them, we would have expected the performance to drop.

(c) This could occur due to various reasons:

i. **Simpler decision boundary**: The hypothesis could probably have enough information in its feature space to make accurate predictions. Thus, **the decision boundary would have been simpler to construct without the noise and complexity from the irrelevant words in the premise** confusing the model.

ii. **Over-fitting or imbalanced features**: The words in the premise could have features with larger values than those in the hypothesis. This would lead to an **imbalance in the feature contributions, especially with irrelevant or informative words in the premise**. The model would become biased towards the premise and over-fit on it with a large part of the prediction score being contributed from the dot product between the weights and the features of the premise. Removing the premise could have led to better generalisation of unseen data, with the dot product being contributed from relevant and distinguishing words in the hypothesis.

Problem 3: Word vectors

In this problem, you will implement functions to compute dense word vectors from a word co-occurrence matrix using SVD, and explore similarities between words. You will be using python packages `nltk` and `numpy` for this problem.
We will estimate word vectors using the corpus *Emma* by Jane Austen from `nltk`. Take a look at the function `read_corpus` in *util.py* which downloads the corpus.

1. [3 points, coding] First, let's construct the word co-occurrence matrix. Implement the function `count_cooccur_matrix` using a window size of 4 (i.e. considering 4 words before and 4 words after the center word).

```python
def count_cooccur_matrix(tokens, window_size=4):
    """Compute the co-occurrence matrix given a sequence of tokens.
    For each word, n words before and n words after it are its co-occurring neighbors.
    For example, given the tokens "in for a penny , in for a pound",
    the neighbors of "penny" given a window size of 2 are "for", "a", ",", "in".
    Parameters:
        tokens : [str]
        window_size : int
    Returns:
        word2ind : dict
            word (str) : index (int)
        co_mat : np.array
            co_mat[i][j] should contain the co-occurrence counts of the words indexed
            by i and j according to the dictionary word2ind.
    """
    vocabulary = set(tokens)
    vocabulary_size = len(vocabulary)
    # compute a dictionary mapping words to indices
    word2ind = {word: idx for idx, word in enumerate(vocabulary)}
    # initialize the co-occurrence matrix
    co_mat = np.zeros((vocabulary_size, vocabulary_size))
    for i, token in enumerate(tokens):
        # the window size comprises tokens in the range
        # [i - window_size, i + window_size], excluding the token itself
        for j in range(max(i - window_size, 0), min(i + window_size + 1, len(tokens))):
            if i != j:
                # increment the count of the co-occurring words
                co_mat[word2ind[token]][word2ind[tokens[j]]] += 1

    return word2ind, co_mat
```

2. [1 points] Next, let's perform dimensionality reduction on the co-occurrence matrix to obtain dense word vectors. You will implement truncated SVD using the `numpy.linalg.svd` function in the next part. Read its documentation (`https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html`) carefully. Can we set `hermitian` to `True` to speed up the computation? Explain your answer in one sentence.

A word-cooccurrence matrix is always **square**, **symmetric** and **real-valued**, i.e, the number of times word w_i appears in the neighbourhood of word w_j is equal to the number of times the word w_j appears in the neighbourhood of word w_i, and thus, obeys **Hermitian characteristics** and this can be leveraged to optimise and speed up computation.

3. [3 points, coding] Now, implement the `cooccur_to_embedding` function that returns word embeddings based on truncated SVD.

```python
def cooccur_to_embedding(co_mat, embed_size=50):
    """Convert the co-occurrence matrix to word embedding using truncated SVD.
    Use the np.linalg.svd function.
    Parameters:
        co_mat : np.array
            vocab size x vocab size
        embed_size : int
    Returns:
        embeddings : np.array
            vocab_size x embed_size
    """
    # compute the SVD of the co-occurrence matrix. We opt for the full SVD here
    # to obtain both U and S matrices.
    # we set hermitian=True because the matrix is real and symmetric
    # (since a co-occurrence matrix is always symmetric)
    u, s, vt = np.linalg.svd(co_mat, full_matrices=True, hermitian=True)
    # we take only the first embed_size columns of U and multiply it with a
    # diagonal matrix of the first embed_size singular values
    embeddings = u[:, :embed_size] @ np.diag(s[:embed_size])
    return embeddings
```

4. [2 points, coding] Let's play with the word embeddings and see what they capture. In particular, we will find the most similar words to a given word. In order to do that, we need to define a similarity function between two word vectors. Dot product is one such metric. Implement it in `top_k_similar` (where `metric='dot'`).

```python
def top_k_similar(word_ind, embeddings, word2ind, k=10, metric="dot"):
    """Return the top k most similar words to the given word (excluding itself).
    You will implement two similarity functions.
    If metric='dot', use the dot product.
    If metric='cosine', use the cosine similarity.
    Parameters:
        word_ind : int
            index of the word (for which we will find the similar words)
        embeddings : np.array
            vocab_size x embed_size
        word2ind : dict
        k : int
            number of words to return (excluding self)
        metric : 'dot' or 'cosine'
    Returns:
        topk-words : [str]
    """
    # compute a dictionary mapping indices to words
    ind2word = {ind: word for word, ind in word2ind.items()}
    query_word_embedding = embeddings[word_ind]
    # compute the similarity scores of the query word with all the words
    similarity_scores = np.array(
        list(map(lambda v: np.dot(query_word_embedding, v), embeddings))
    )
    # sort the similarity scores in descending order and return the top k words
    # we exclude the query word itself by starting the slicing from the second element
    top_k_indices = similarity_scores.argsort()[::-1][1 : k + 1]
    return list(map(lambda x: ind2word[x], top_k_indices))
```

5. [1 points] Now, run `test_embedding.py` to get the top-k words. What's your observation? Explain why that is the case.

(a) We observe that the top-k words have returned irrelevant words, **primarily comprising stop words** such as "i", "the", and "it" and other **high-frequency words**. Some query words like "sad" also return zero relevant results.

(b) We also observe that in some cases (such as the word "knightley"), the word itself is not the most relevant. This is because words with higher frequencies (such as stop words) also have a higher co-occurrence count value in the word co-occurrence matrix. The **larger magnitude of the high-frequency or stop word vector embedding dominates the similarity computation** with the query word's vector embedding, resulting in the dot product being larger than a relevant word's dot product with the same query word.

(c) Since we are sorting and returning the words with the highest dot product, **the algorithm returns the most commonly occurring words around a word**. This can be handled by normalising the word vectors to handle the high magnitude of some high-frequency words.

6. [2 points, coding] To fix the issue, implement the cosine similarity function in `top_k_similar` (where `metric='cosine'`).

```python
def top_k_similar(word_ind, embeddings, word2ind, k=10, metric="dot"):
    """Return the top k most similar words to the given word (excluding itself).
    You will implement two similarity functions.
    If metric='dot', use the dot product.
    If metric='cosine', use the cosine similarity.
    Parameters:
        word_ind : int
            index of the word (for which we will find the similar words)
        embeddings : np.array
            vocab_size x embed_size
        word2ind : dict
        k : int
            number of words to return (excluding self)
        metric : 'dot' or 'cosine'
    Returns:
        topk-words : [str]
    """
    # compute a dictionary mapping indices to words
    ind2word = {ind: word for word, ind in word2ind.items()}
    # normalize the embeddings for quicker computation
    embeddings /= np.linalg.norm(embeddings, axis=1, keepdims=True)
    query_word_embedding = embeddings[word_ind]
    # compute the similarity scores of the query word with all the words
    # since we have normalized the embeddings, the cosine similarity is equivalent
    # to the dot product
    similarity_scores = np.array(
        list(map(lambda v: np.dot(query_word_embedding, v), embeddings))
    )
    # sort the similarity scores in descending order and return the top k words
    # we exclude the query word itself by starting the slicing from the second element
    top_k_indices = similarity_scores.argsort()[::-1][1 : k + 1]
    return list(map(lambda x: ind2word[x], top_k_indices))
```

7. [1 points] Among the given word list, take a look at the top-k similar words of “man” and “woman”, in particular the adjectives. How do they differ? Explain what makes sense and what is surprising.

(a) We observe that the top-k similar words for the word ”man” comprises words such as ”gallant”, ”fine” and ”charming”, which are often roles/characteristics considered to be masculine. Similarly, the top-k similar words for the word ”woman” comprises words such as ”sweet”, ”blush” and ”pert”, which are roles/characteristics usually considered to be feminine. Thus, this **indicates the presence of gender stereotypes and biases in the training data**, which are very harmful in real-world applications. Since these stereotypes were present in the training data, the embedding model has also preserved these stereotypes in its word embeddings.

(b) However, we also observe a tiny overlap in the top-k words namely ”charming”, ”farmer”, ”fine” and ”pert”, which could indicate that **some characteristics cannot be definitively classified** for each gender from the provided data. The most similar word to ”man” being ”woman” and vice versa could indicate that the training data contained a **large amount of text that correlated the concepts** of man and woman together in great detail.

(c) It is surprising to observe the word ”weak” for the word ”man” and ”farmer” for the word ”woman” since men are stereotypically considered to be strong and farming is usually considered a masculine occupation. This indicates that the word ”weak” was **commonly related to discussions of men** in the training data. The occurrence of the word ”farmer” across both genders with a high similarity score indicates that it was **very commonly used in discussions with both genders** and that the training data could be related to farming and consider it to be a pivotal role/characteristic for both genders.

8. [1 points] Among the given word list, take a look at the top-k similar words of “happy” and “sad”. Do they contain mostly synonyms, or antonyms, or both? What do you expect and why?

(a) We can find **both synonyms and antonyms** for both "happy" and "sad" in their respective result sets. The result set for the word "happy" comprises words such as "pleasant" and "agreeable" as well as words such as "distressing and "unpleasant", while the result set for the word "sad" comprises words such as "quivering" and "lame" as well as "delightful".

(b) The **expected behaviour would have been to observe *only* synonyms** because semantically similar words (synonyms) should ideally display higher similarity scores compared to antonyms. Ideally, synonyms and antonyms are semantically opposite concepts and the **cosine similarity score *should have* been highly positive for synonyms and highly negative for antonyms**.

(c) The presence of both synonyms and antonyms implies that **the training data could include sentences where both sentiments appear frequently together**. This would lead to the words "happy" and "sad" being contextually related, i.e. they are either used in contrast with one another in a sentence or their association with synonyms or antonyms in a sentence is very common.