CGOE

Pset 4 – Office Hours

1. **KL divergence and Maximum Likelihood**

(a) **[5 points (Written)] Nonnegativity.**

Prove the following:

$$\forall P, Q. \quad D_{KL}(P \,||\, Q) \geq 0$$

and

$$D_{KL}(P \,||\, Q) = 0 \quad \boxed{\text{if and only if}} \quad P = Q.$$

Prove in both directions

[Hint: You may use the following result, called **Jensen's inequality**. If f is a convex function, and X is a random variable, then $E[f(X)] \geq f(E[X])$. Moreover, if f is strictly convex (f is convex if its Hessian satisfies $H \geq 0$; it is *strictly* convex if $H > 0$; for instance $f(x) = -\log x$ is strictly convex), then $E[f(X)] = f(E[X])$ implies that $X = E[X]$ with probability 1; i.e., X is actually a constant.]

Hints:

1. Start with $-D_{KL}(P \,||\, Q)$ and prove = 0
2. Use def of KL divergence
3. Recall $-\log\left(\frac{a}{b}\right) = \log\left(\frac{b}{a}\right)$
4. Jensen's inequality:
    1. recall def of expectation: $\boldsymbol{E}[X] = \sum x_i p_i$
    2. recognize that $\log(t)$ is convex for any t

(b) **[4 points (Written)] Chain rule for KL divergence.**

The KL divergence between 2 conditional distributions $P(X \mid Y), Q(X \mid Y)$ is defined as follows:

$$D_{KL}(P(X \mid Y) \,||\, Q(X \mid Y)) = \sum_y P(y) \left(\sum_x P(x \mid y) \log \frac{P(x \mid y)}{Q(x \mid y)} \right)$$

This can be thought of as the expected KL divergence between the corresponding conditional distributions on x (that is, between $P(X \mid Y = y)$ and $Q(X \mid Y = y)$), where the expectation is taken over the random y.

Prove the following chain rule for KL divergence:

$$D_{KL}(P(X,Y) \,||\, Q(X,Y)) = D_{KL}(P(X) \,||\, Q(X)) + D_{KL}(P(Y \mid X) \,||\, Q(Y \mid X)).$$

Hints:

1. Start with LHS □ RHS
2. Substitute in the def of KL convergence
3. Recall the definition of conditional probability

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

(c) [4 points (Written)] **KL and maximum likelihood.**

Consider a density estimation problem, and suppose we are given a training set $\{x^{(i)}; i = 1, \ldots, n\}$. Let the empirical distribution be $\hat{P}(x) = \frac{1}{n}\sum_{i=1}^{n} 1\{x^{(i)} = x\}$. ($\hat{P}$ is just the uniform distribution over the training set; i.e., sampling from the empirical distribution is the same as picking a random example from the training set.)

Suppose we have some family of distributions P_θ parameterized by θ. (If you like, think of $P_\theta(x)$ as an alternative notation for $P(x;\theta)$.) Prove that finding the maximum likelihood estimate for the parameter θ is equivalent to finding P_θ with minimal KL divergence from $\hat{P}$. I.e. prove:

$$\arg\min_\theta D_{KL}(\hat{P} \,||\, P_\theta) = \arg\max_\theta \sum_{i=1}^{n} \log P_\theta(x^{(i)})$$

Remark. Consider the relationship between parts (b-c) and multi-variate Bernoulli Naive Bayes parameter estimation. In the Naive Bayes model we assumed P_θ is of the following form: $P_\theta(x, y) = p(y)\prod_{i=1}^{d} p(x_i \mid y)$. By the chain rule for KL divergence, we therefore have:

$$D_{KL}(\hat{P} \,||\, P_\theta) = D_{KL}(\hat{P}(y) \,||\, p(y)) + \sum_{i=1}^{d} D_{KL}(\hat{P}(x_i \mid y) \,||\, p(x_i \mid y)).$$

This shows that finding the maximum likelihood/minimum KL-divergence estimate of the parameters decomposes into $2n + 1$ independent optimization problems: One for the class priors $p(y)$, and one for each of the conditional distributions $p(x_i \mid y)$ for each feature x_i given each of the two possible labels for y. Specifically, finding the maximum likelihood estimates for each of these problems individually results in also maximizing the likelihood of the joint distribution. (If you know what Bayesian networks are, a similar remark applies to parameter estimation for them.)

(c) **[4 points (Written)] KL and maximum likelihood.**

Consider a density estimation problem, and suppose we are given a training set $\{x^{(i)}; i = 1, \ldots, n\}$. Let the empirical distribution be $\hat{P}(x) = \frac{1}{n}\sum_{i=1}^{n} 1\{x^{(i)} = x\}$. ($\hat{P}$ is just the uniform distribution over the training set; i.e., sampling from the empirical distribution is the same as picking a random example from the training set.)

Suppose we have some family of distributions P_θ parameterized by θ. (If you like, think of $P_\theta(x)$ as an alternative notation for $P(x;\theta)$.) Prove that finding the maximum likelihood estimate for the parameter θ is equivalent to finding P_θ with minimal KL divergence from $\hat{P}$. I.e. prove:

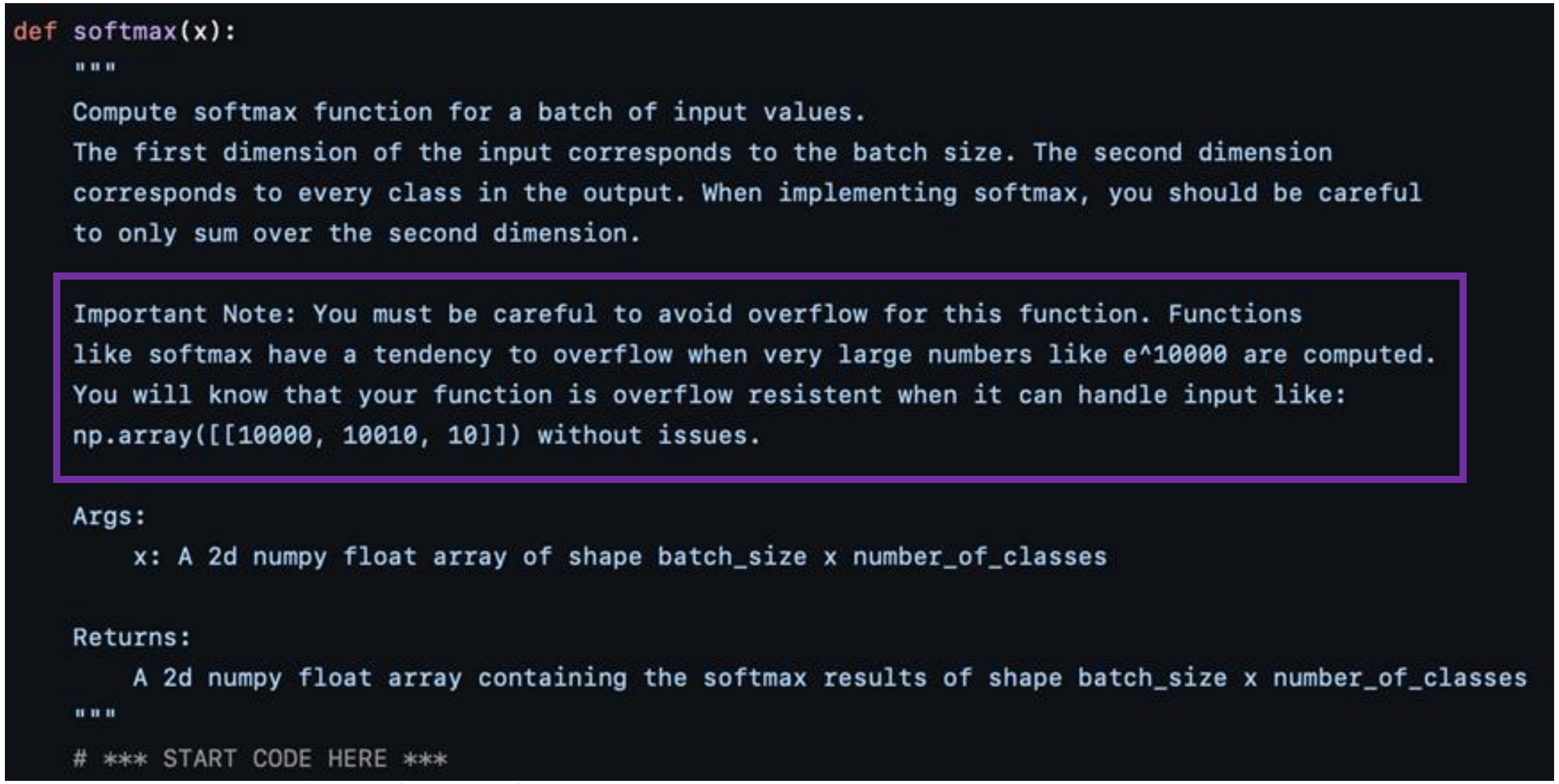
$$\arg\min_\theta D_{KL}(\hat{P} \,||\, P_\theta) = \arg\max_\theta \sum_{i=1}^{n} \log P_\theta(x^{(i)})$$

Hints:

1. Start with LHS □ RHS
2. Substitute in the def of KL convergence
3. Use argmax and argmin to eliminate terms that do not depend on arg
4. The arg max of a term = the arg min of its negative
5. The order of summations can be changed

2. Neural Networks: Fashion-MNIST image classification

coding
:)

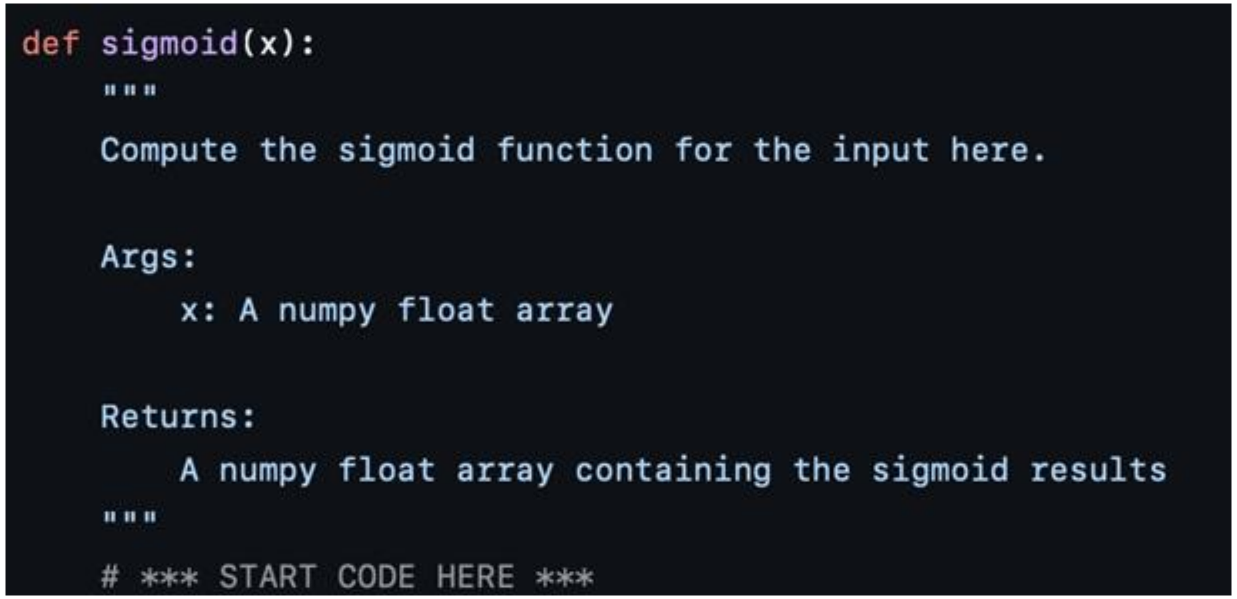


```python
def softmax(x):
    """
    Compute softmax function for a batch of input values.
    The first dimension of the input corresponds to the batch size. The second dimension
    corresponds to every class in the output. When implementing softmax, you should be careful
    to only sum over the second dimension.

    Important Note: You must be careful to avoid overflow for this function. Functions
    like softmax have a tendency to overflow when very large numbers like e^10000 are computed.
    You will know that your function is overflow resistent when it can handle input like:
    np.array([[10000, 10010, 10]]) without issues.

    Args:
        x: A 2d numpy float array of shape batch_size x number_of_classes

    Returns:
        A 2d numpy float array containing the softmax results of shape batch_size x number_of_classes
    """
    # *** START CODE HERE ***
```

Not going to use a 'trivial' implementation of softmax

our solution is 4 lines of code

$$\text{softmax}(t_1, \ldots, t_k) = \begin{bmatrix} \frac{\exp(t_1)}{\sum_{j=1}^{k} \exp(t_j)} \\ \vdots \\ \frac{\exp(t_k)}{\sum_{j=1}^{k} \exp(t_j)} \end{bmatrix} \longrightarrow \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{-m}}{e^{-m}} \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - m}}{\sum_j e^{x_j - m}}$$



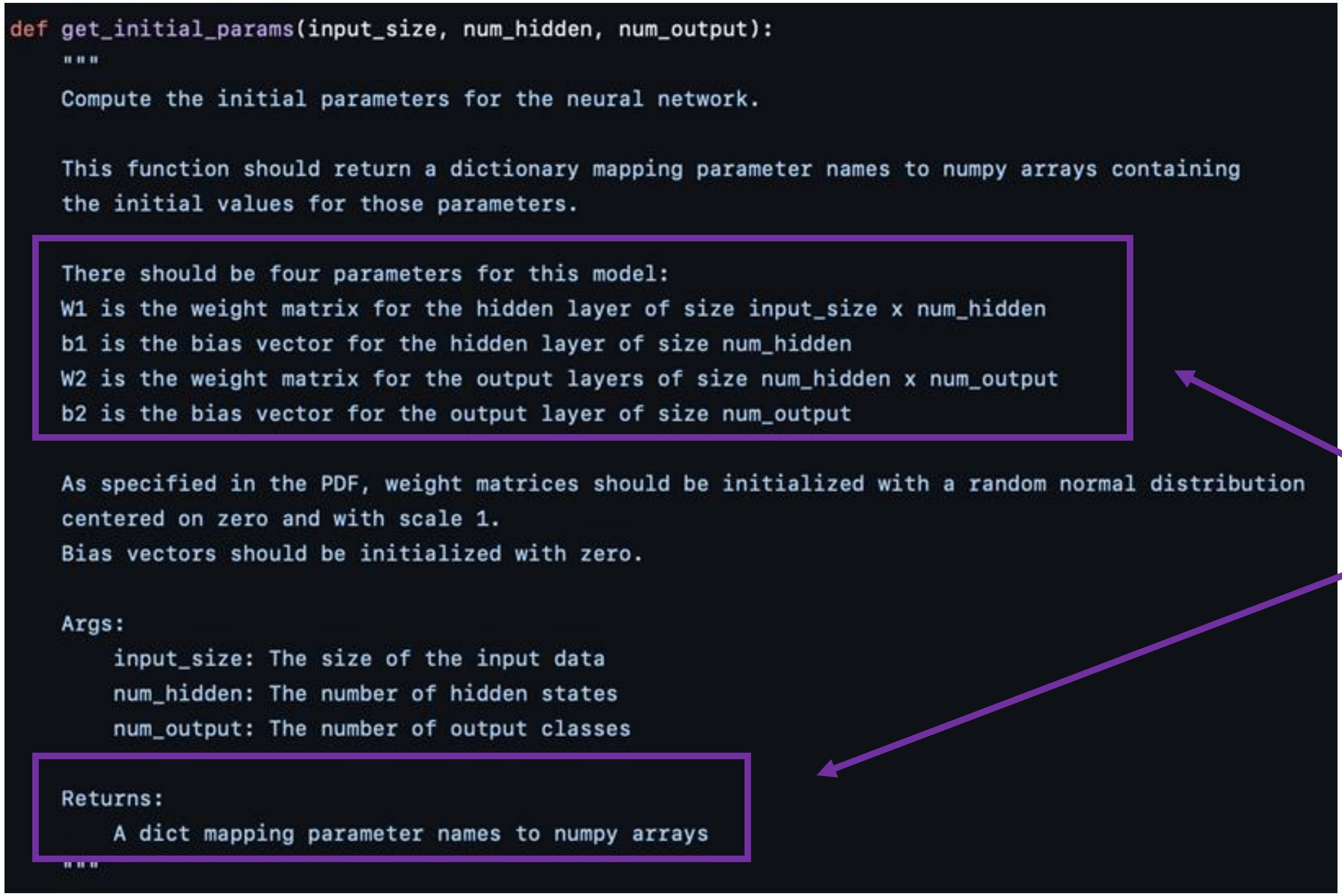
```python
def sigmoid(x):
    """
    Compute the sigmoid function for the input here.

    Args:
        x: A numpy float array

    Returns:
        A numpy float array containing the sigmoid results
    """
    # *** START CODE HERE ***
```

Will also need to prevent over / under flowing!!!

our solution is 4 lines of code

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^{x}}{1+e^{x}}$$

One of these equations prevents issues if x is large negative and the other if large positive – you figure out how to implement



```
def get_initial_params(input_size, num_hidden, num_output):
    """
    Compute the initial parameters for the neural network.

    This function should return a dictionary mapping parameter names to numpy arrays containing
    the initial values for those parameters.

    There should be four parameters for this model:
    W1 is the weight matrix for the hidden layer of size input_size x num_hidden
    b1 is the bias vector for the hidden layer of size num_hidden
    W2 is the weight matrix for the output layers of size num_hidden x num_output
    b2 is the bias vector for the output layer of size num_output

    As specified in the PDF, weight matrices should be initialized with a random normal distribution
    centered on zero and with scale 1.
    Bias vectors should be initialized with zero.

    Args:
        input_size: The size of the input data
        num_hidden: The number of hidden states
        num_output: The number of output classes

    Returns:
        A dict mapping parameter names to numpy arrays
    """
```

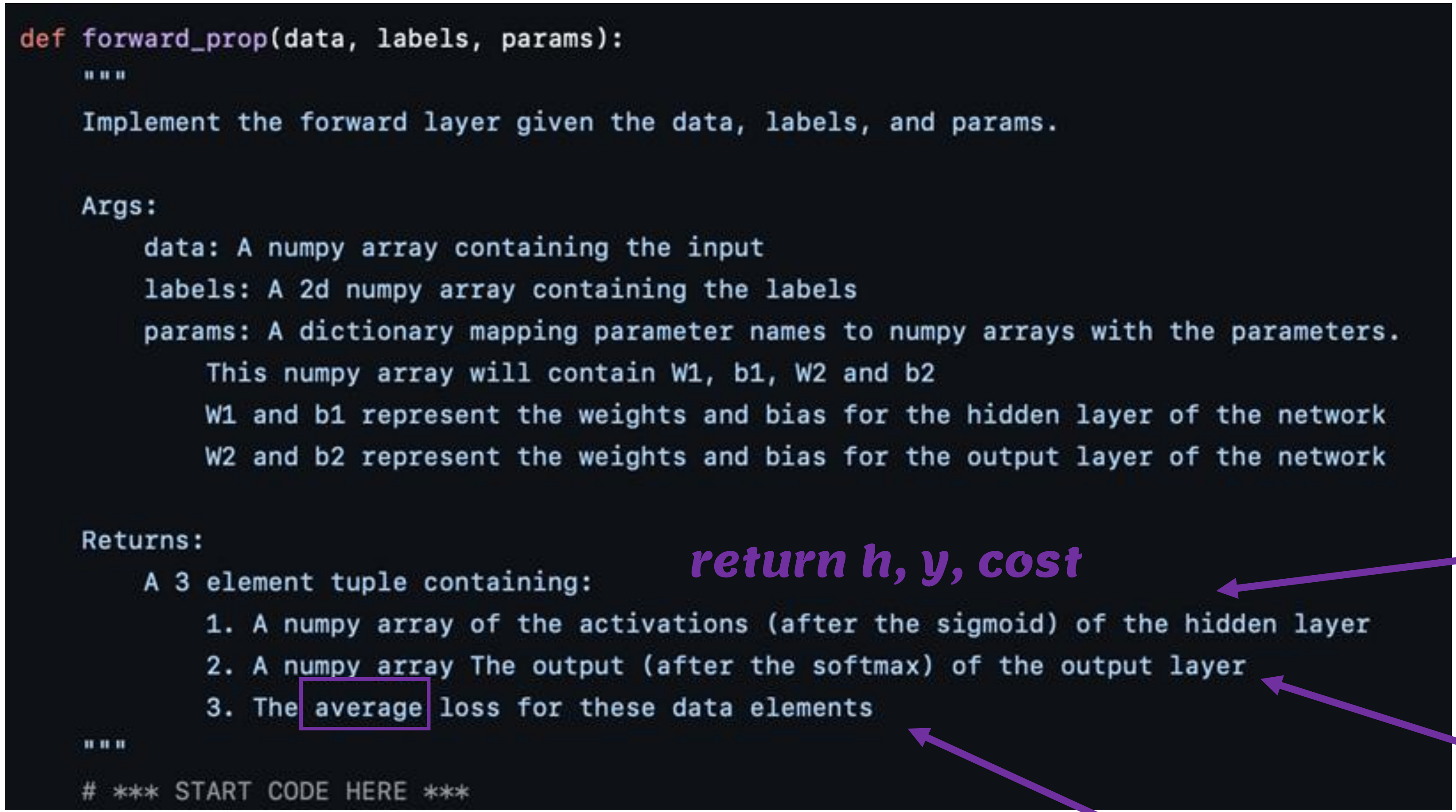
Return type:

Dict[str, np array]

np.random.normal is a helpful function

Check shapes!

our solution is ~6 lines of code



```
def forward_prop(data, labels, params):
    """
    Implement the forward layer given the data, labels, and params.

    Args:
        data: A numpy array containing the input
        labels: A 2d numpy array containing the labels
        params: A dictionary mapping parameter names to numpy arrays with the parameters.
            This numpy array will contain W1, b1, W2 and b2
            W1 and b1 represent the weights and bias for the hidden layer of the network
            W2 and b2 represent the weights and bias for the output layer of the network

    Returns:
        A 3 element tuple containing:
            1. A numpy array of the activations (after the sigmoid) of the hidden layer
            2. A numpy array The output (after the softmax) of the output layer
            3. The average loss for these data elements
    """
    # *** START CODE HERE ***
```

return h, y, cost

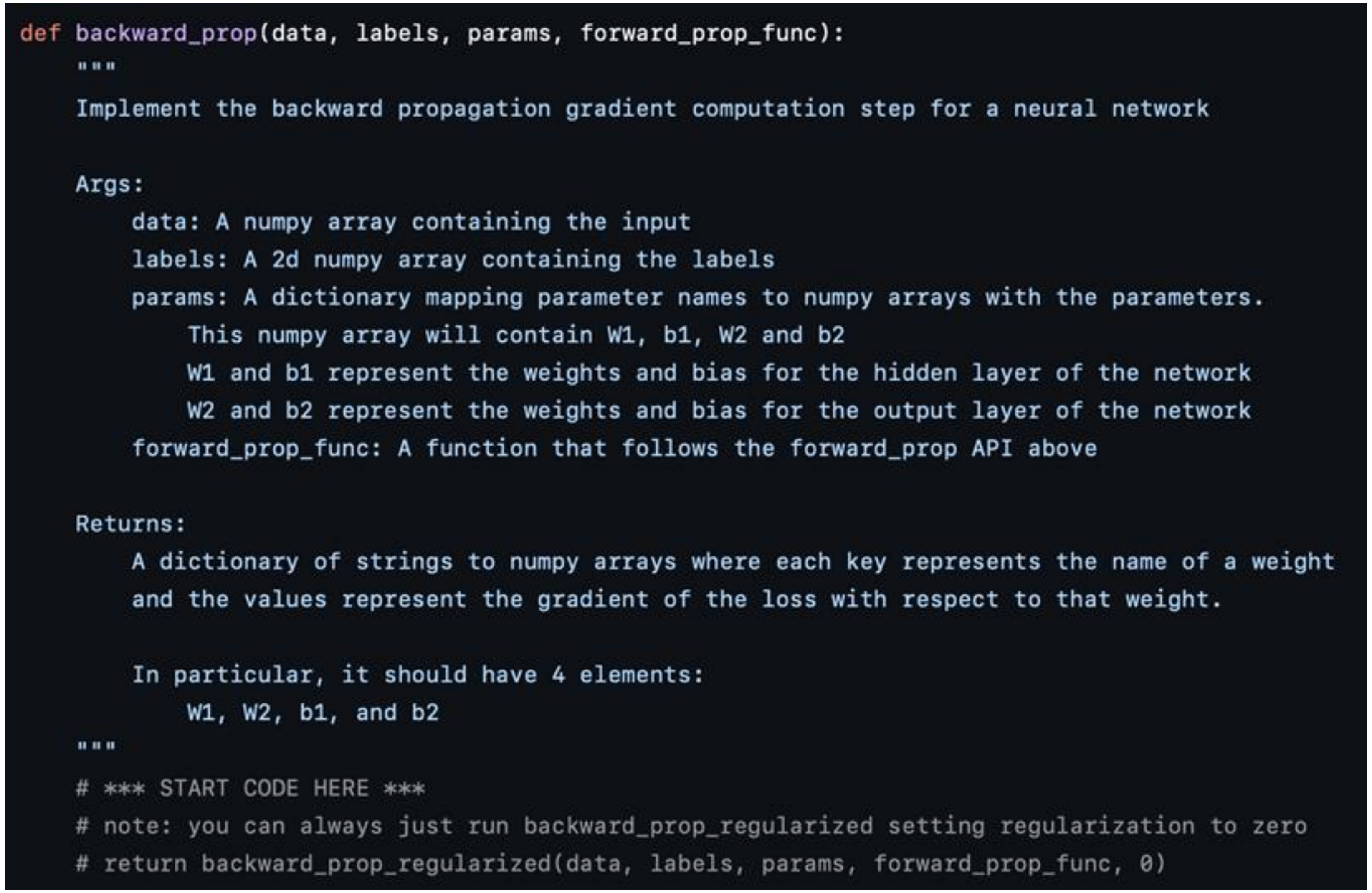
$$a^{(i)} = \sigma\left({W^{[1]}}^\top x^{(i)} + b^{[1]}\right)$$

$$z^{(i)} = {W^{[2]}}^\top a^{(i)} + b^{[2]}$$

$$\hat{y}^{(i)} = \text{softmax}(z^{(i)})$$

our solution is 8 lines of code

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^{n} CE(y^{(i)}, \hat{y}^{(i)}) = \boxed{-\frac{1}{n}}\sum_{i=1}^{n}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}$$



```python
def backward_prop(data, labels, params, forward_prop_func):
    """
    Implement the backward propagation gradient computation step for a neural network

    Args:
        data: A numpy array containing the input
        labels: A 2d numpy array containing the labels
        params: A dictionary mapping parameter names to numpy arrays with the parameters.
            This numpy array will contain W1, b1, W2 and b2
            W1 and b1 represent the weights and bias for the hidden layer of the network
            W2 and b2 represent the weights and bias for the output layer of the network
        forward_prop_func: A function that follows the forward_prop API above

    Returns:
        A dictionary of strings to numpy arrays where each key represents the name of a weight
        and the values represent the gradient of the loss with respect to that weight.

        In particular, it should have 4 elements:
            W1, W2, b1, and b2
    """
    # *** START CODE HERE ***
    # note: you can always just run backward_prop_regularized setting regularization to zero
    # return backward_prop_regularized(data, labels, params, forward_prop_func, 0)
```

our solution is 15 lines of code

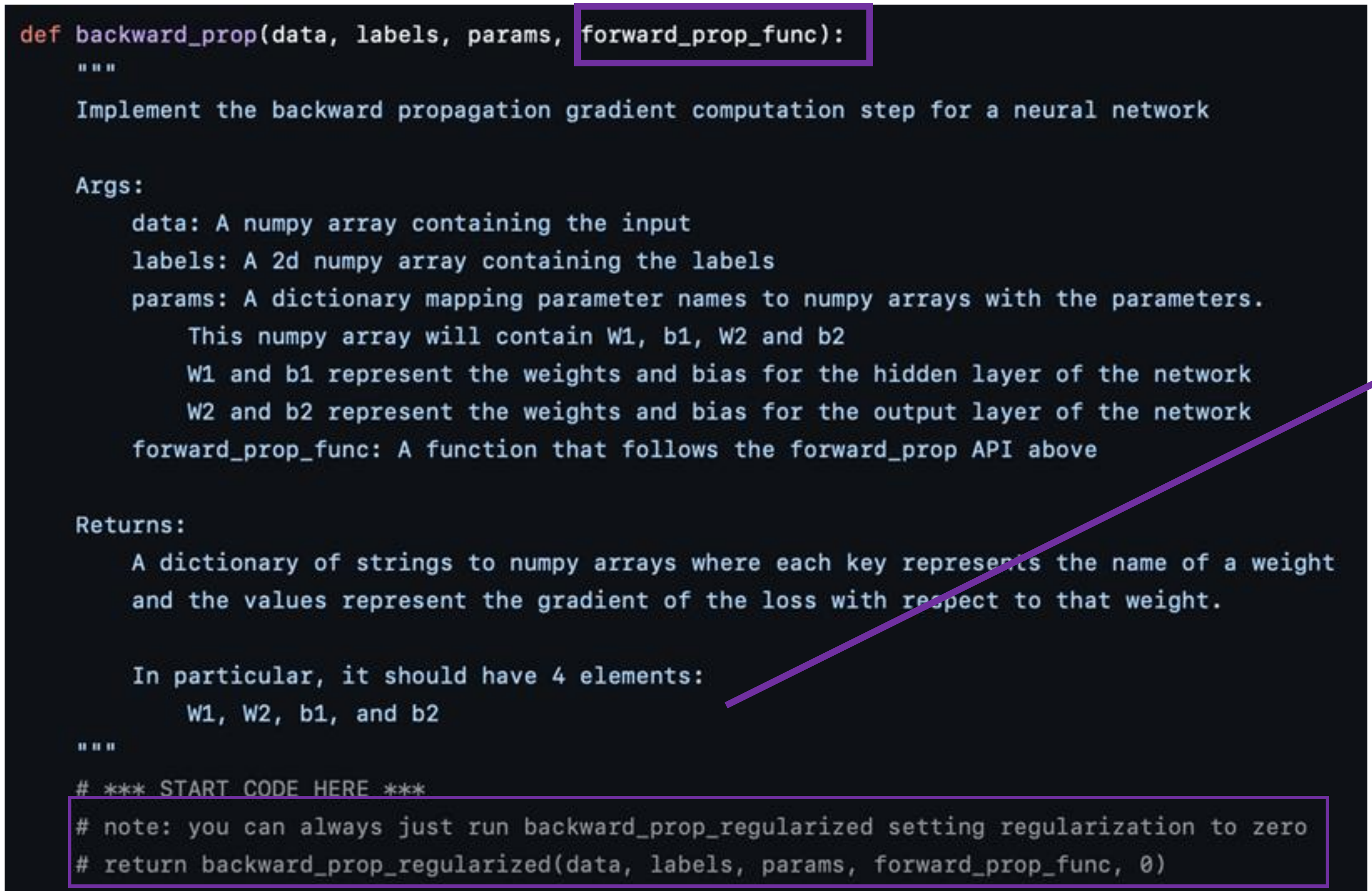
$$\frac{dJ}{dW_2} = \boxed{\frac{dJ}{dZ}} \frac{dZ}{dW_2}$$

$$\frac{dJ}{db_2} = \boxed{\frac{dJ}{dZ}} \frac{dZ}{db_2}$$

$$\frac{dJ}{dW_1} = \boxed{\frac{dJ}{dZ}} \frac{dZ}{dA} \frac{dA}{dW_1}$$

$$\frac{dJ}{db_1} = \boxed{\frac{dJ}{dZ}} \frac{dZ}{dA} \frac{dA}{db_1}$$

$$\frac{\partial J}{\partial Z} = Y_{hat} - Y$$



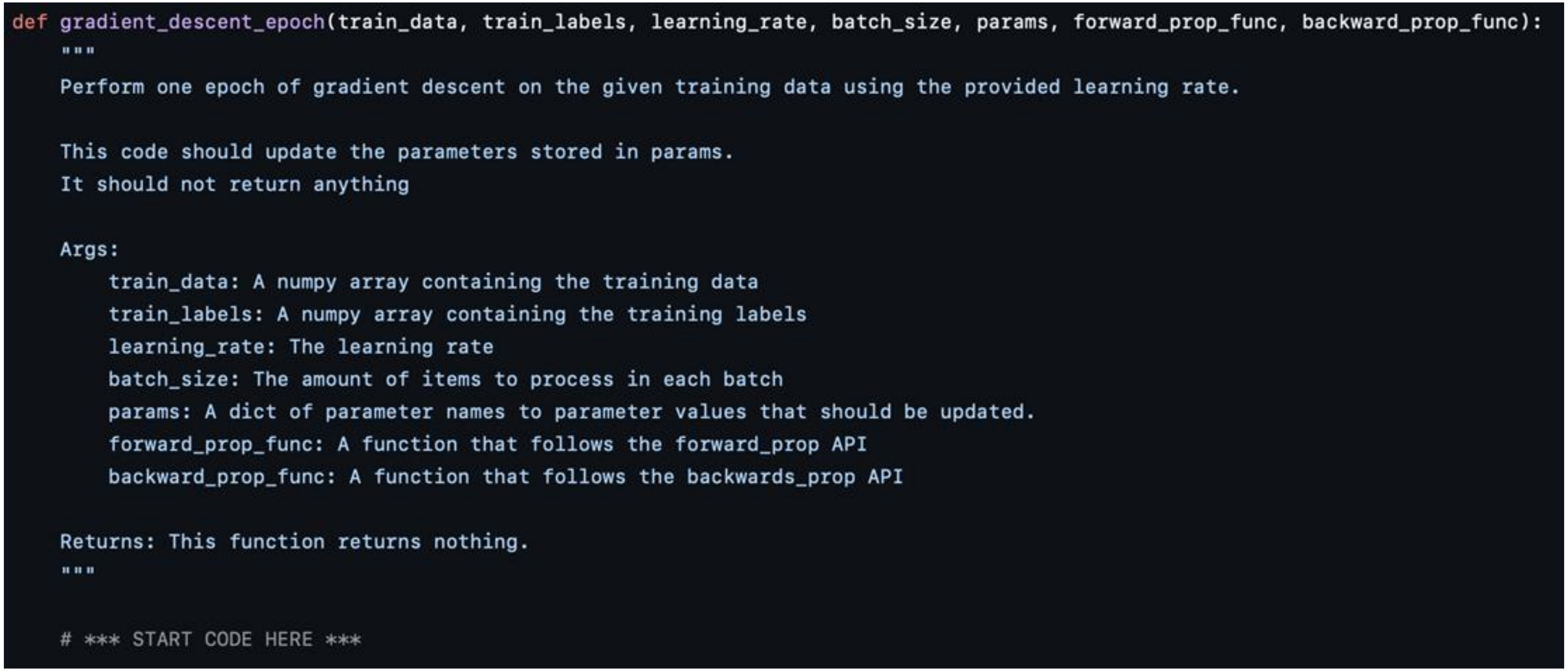
```python
def backward_prop(data, labels, params, forward_prop_func):
    """
    Implement the backward propagation gradient computation step for a neural network

    Args:
        data: A numpy array containing the input
        labels: A 2d numpy array containing the labels
        params: A dictionary mapping parameter names to numpy arrays with the parameters.
            This numpy array will contain W1, b1, W2 and b2
            W1 and b1 represent the weights and bias for the hidden layer of the network
            W2 and b2 represent the weights and bias for the output layer of the network
        forward_prop_func: A function that follows the forward_prop API above

    Returns:
        A dictionary of strings to numpy arrays where each key represents the name of a weight
        and the values represent the gradient of the loss with respect to that weight.

        In particular, it should have 4 elements:
            W1, W2, b1, and b2
    """
    # *** START CODE HERE ***
    # note: you can always just run backward_prop_regularized setting regularization to zero
    # return backward_prop_regularized(data, labels, params, forward_prop_func, 0)
```

Same keys, shapes as the ***params***

our solution is 15 lines of code

h, y, cost = forward_prop_func(data, labels, params)

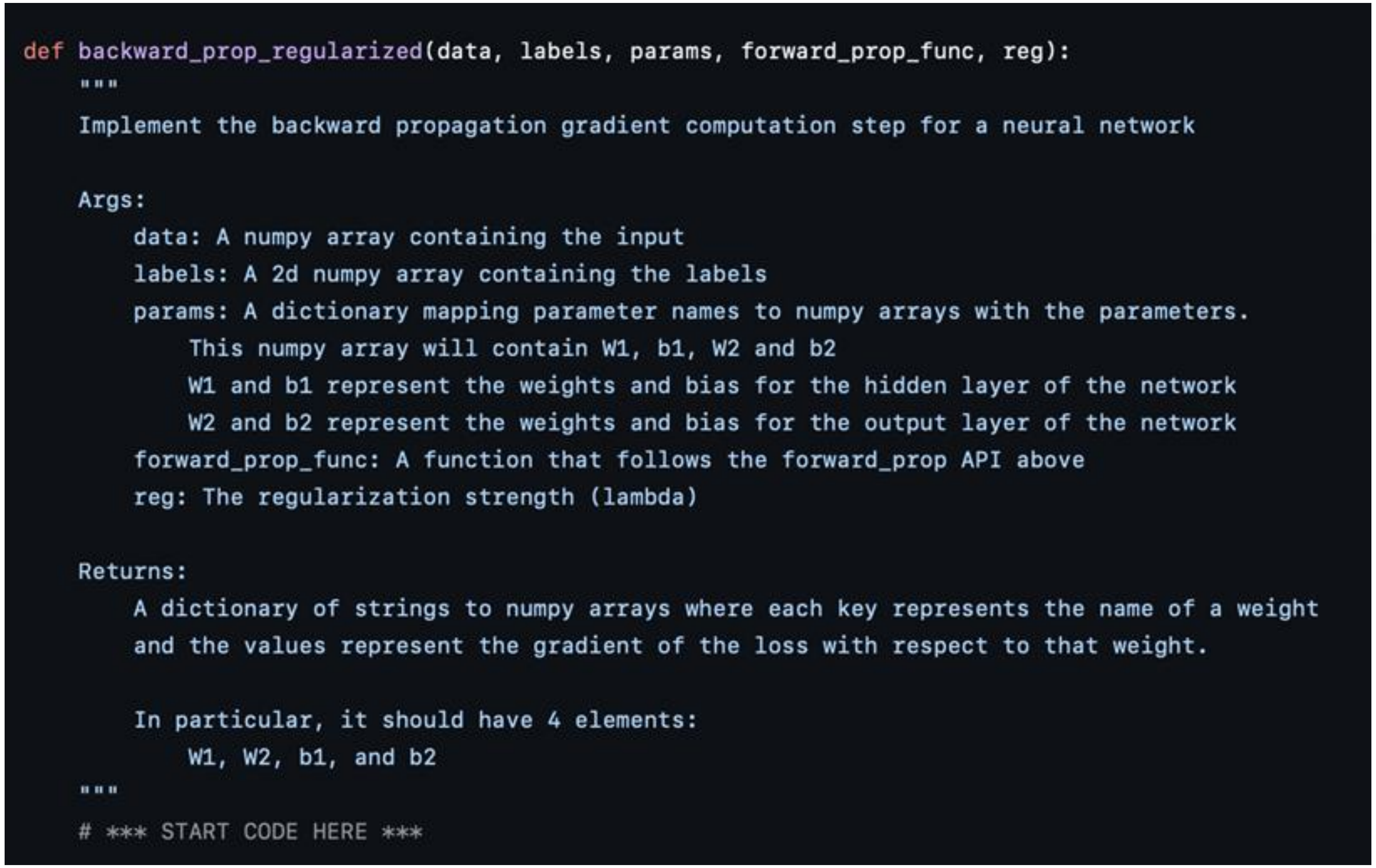
VECTORIZE



```
def gradient_descent_epoch(train_data, train_labels, learning_rate, batch_size, params, forward_prop_func, backward_prop_func):
    """
    Perform one epoch of gradient descent on the given training data using the provided learning rate.

    This code should update the parameters stored in params.
    It should not return anything

    Args:
        train_data: A numpy array containing the training data
        train_labels: A numpy array containing the training labels
        learning_rate: The learning rate
        batch_size: The amount of items to process in each batch
        params: A dict of parameter names to parameter values that should be updated.
        forward_prop_func: A function that follows the forward_prop API
        backward_prop_func: A function that follows the backwards_prop API

    Returns: This function returns nothing.
    """

    # *** START CODE HERE ***
```

our solution is 10 lines of code

Recall that in the code, the forward prop stage is being handled in the back prop function.
All you need to do is implement batches



```python
def backward_prop_regularized(data, labels, params, forward_prop_func, reg):
    """
    Implement the backward propagation gradient computation step for a neural network

    Args:
        data: A numpy array containing the input
        labels: A 2d numpy array containing the labels
        params: A dictionary mapping parameter names to numpy arrays with the parameters.
            This numpy array will contain W1, b1, W2 and b2
            W1 and b1 represent the weights and bias for the hidden layer of the network
            W2 and b2 represent the weights and bias for the output layer of the network
        forward_prop_func: A function that follows the forward_prop API above
        reg: The regularization strength (lambda)

    Returns:
        A dictionary of strings to numpy arrays where each key represents the name of a weight
        and the values represent the gradient of the loss with respect to that weight.

        In particular, it should have 4 elements:
            W1, W2, b1, and b2
    """
    # *** START CODE HERE ***
```

$$\frac{dJ}{dW_2} = \frac{dJ}{dZ}\frac{dZ}{dW_2}$$

$$\frac{dJ}{db_2} = \frac{dJ}{dZ}\frac{dZ}{db_2}$$

$$\frac{dJ}{dW_1} = \frac{dJ}{dZ}\frac{dZ}{dA}\frac{dA}{dW_1}$$

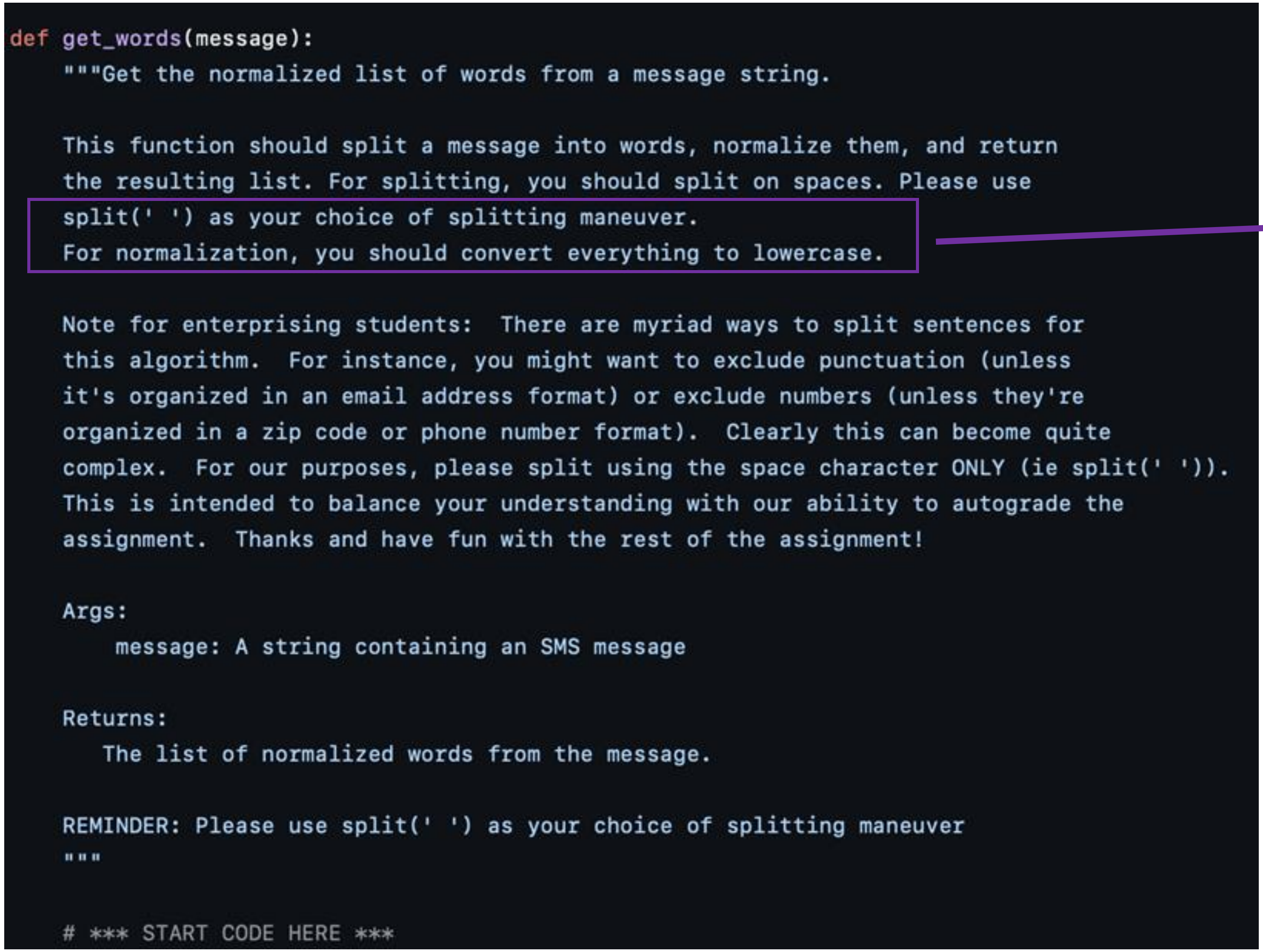
$$\frac{dJ}{db_1} = \frac{dJ}{dZ}\frac{dZ}{dA}\frac{dA}{db_1}$$

$$J_{MB} = \left(\frac{1}{B}\sum_{i=1}^{B} CE(y^{(i)}, \hat{y}^{(i)})\right) + \frac{1}{2}\lambda\left(||W^{[1]}||^2 + ||W^{[2]}||^2\right)$$

Should be copy-paste for the most part from non-reg

3. Spam classification

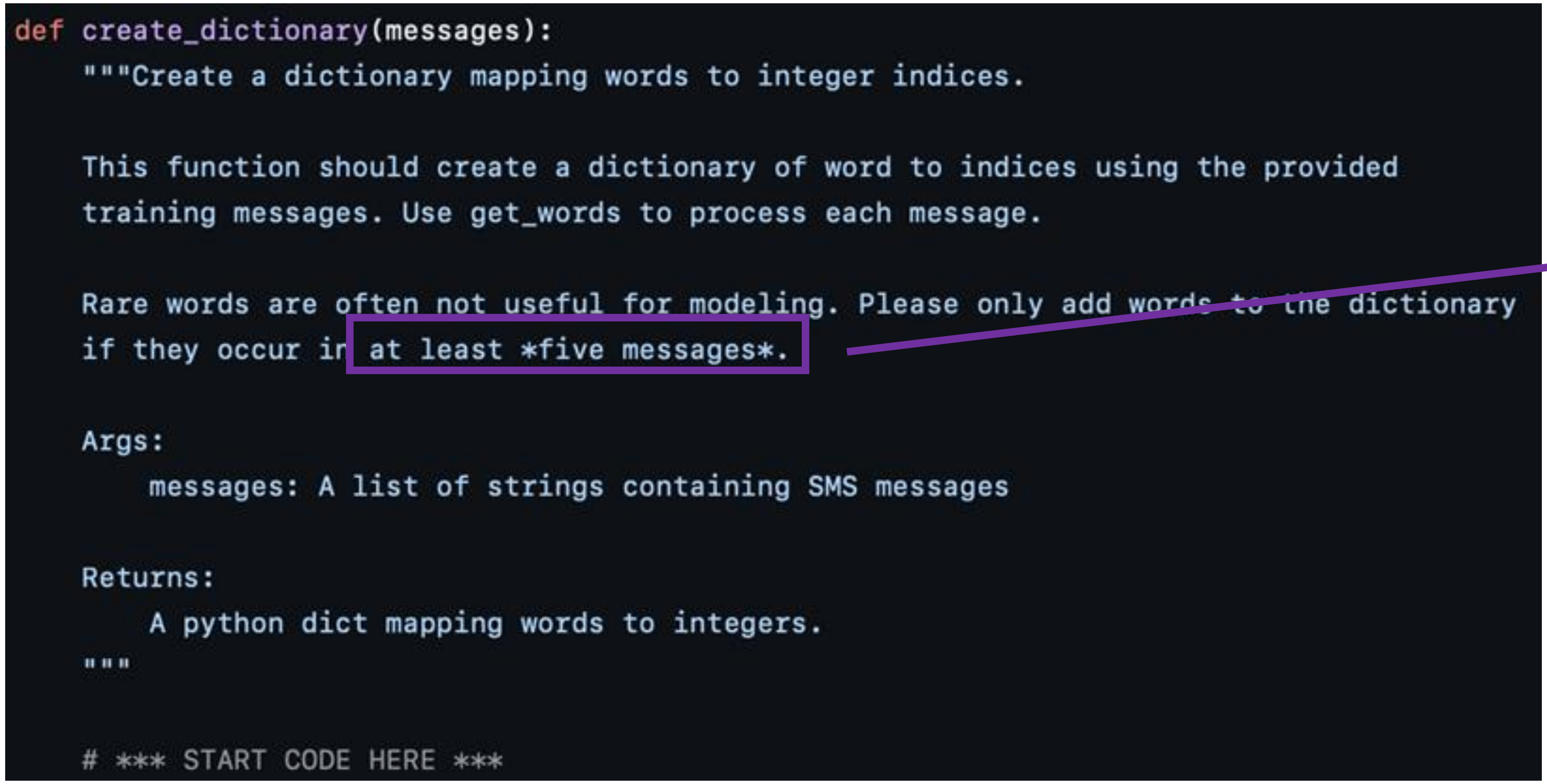
coding
:)



```
def get_words(message):
    """Get the normalized list of words from a message string.

    This function should split a message into words, normalize them, and return
    the resulting list. For splitting, you should split on spaces. Please use
    split(' ') as your choice of splitting maneuver.
    For normalization, you should convert everything to lowercase.

    Note for enterprising students:  There are myriad ways to split sentences for
    this algorithm.  For instance, you might want to exclude punctuation (unless
    it's organized in an email address format) or exclude numbers (unless they're
    organized in a zip code or phone number format).  Clearly this can become quite
    complex.  For our purposes, please split using the space character ONLY (ie split(' ')).
    This is intended to balance your understanding with our ability to autograde the
    assignment.  Thanks and have fun with the rest of the assignment!

    Args:
        message: A string containing an SMS message

    Returns:
       The list of normalized words from the message.

    REMINDER: Please use split(' ') as your choice of splitting maneuver
    """

    # *** START CODE HERE ***
```

1. Use ***split(' ')***
2. Convert to lowercasae

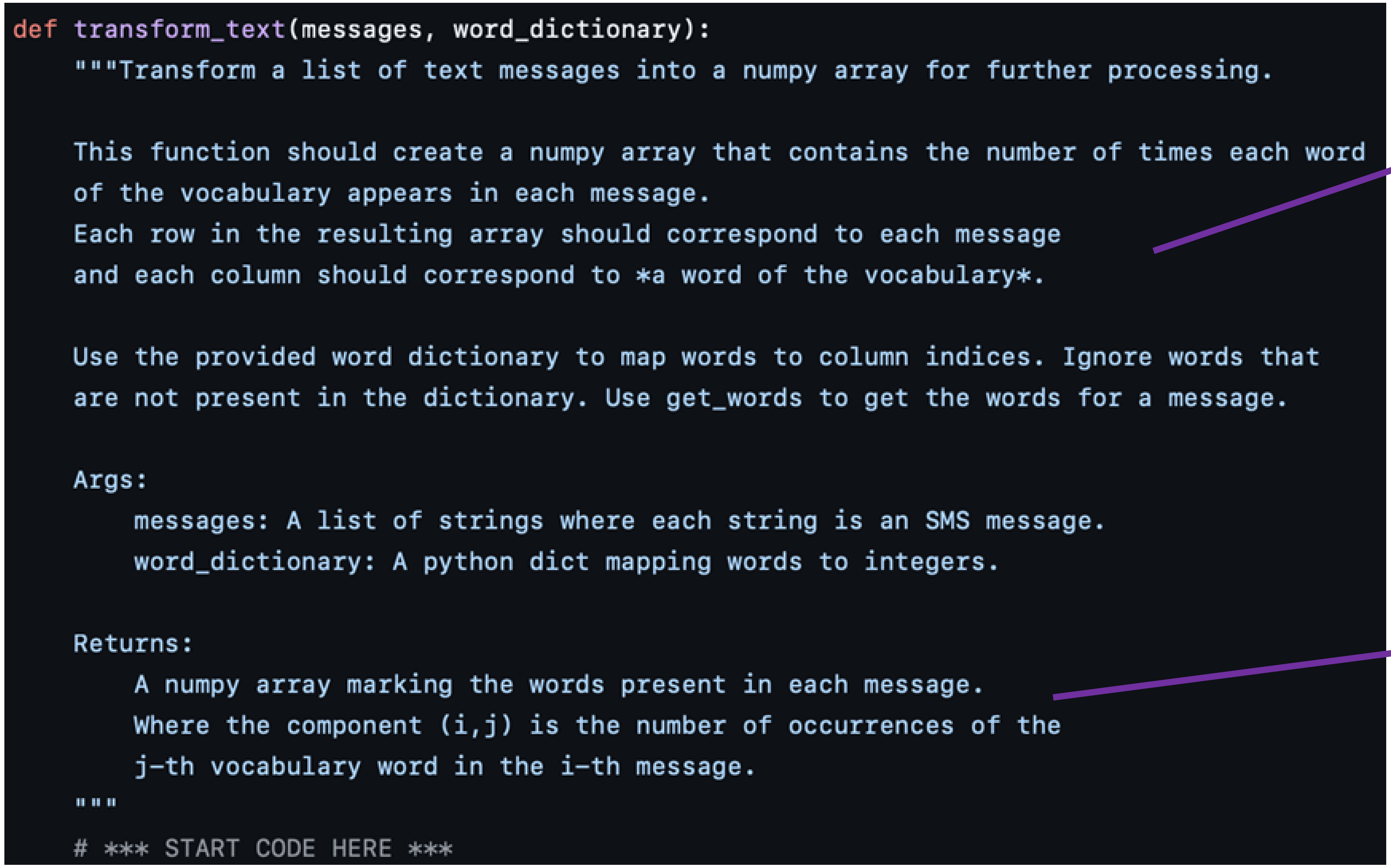
our solution is 1 line of code



```
def create_dictionary(messages):
    """Create a dictionary mapping words to integer indices.

    This function should create a dictionary of word to indices using the provided
    training messages. Use get_words to process each message.

    Rare words are often not useful for modeling. Please only add words to the dictionary
    if they occur in at least *five messages*.

    Args:
        messages: A list of strings containing SMS messages

    Returns:
        A python dict mapping words to integers.
    """

    # *** START CODE HERE ***
```

Five *different* messages

our solution is 10 lines of code

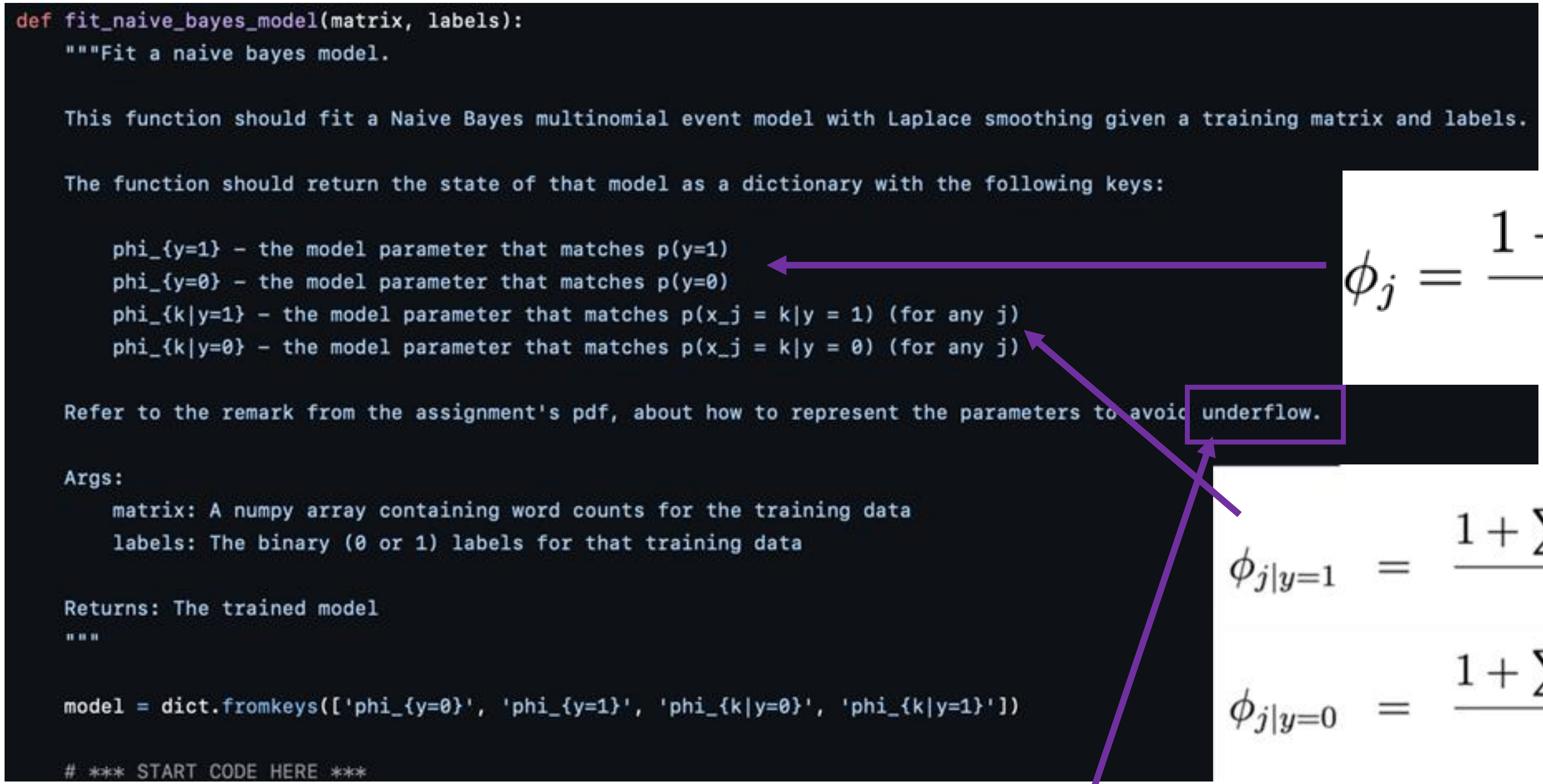


```python
def transform_text(messages, word_dictionary):
    """Transform a list of text messages into a numpy array for further processing.

    This function should create a numpy array that contains the number of times each word
    of the vocabulary appears in each message.
    Each row in the resulting array should correspond to each message
    and each column should correspond to *a word of the vocabulary*.

    Use the provided word dictionary to map words to column indices. Ignore words that
    are not present in the dictionary. Use get_words to get the words for a message.

    Args:
        messages: A list of strings where each string is an SMS message.
        word_dictionary: A python dict mapping words to integers.

    Returns:
        A numpy array marking the words present in each message.
        Where the component (i,j) is the number of occurrences of the
        j-th vocabulary word in the i-th message.
    """
    # *** START CODE HERE ***
```

Big numpy array of messages x words

Count the occurrences of each word in every message

our solution is 6 lines of code



```
def fit_naive_bayes_model(matrix, labels):
    """Fit a naive bayes model.

    This function should fit a Naive Bayes multinomial event model with Laplace smoothing given a training matrix and labels.

    The function should return the state of that model as a dictionary with the following keys:

        phi_{y=1} - the model parameter that matches p(y=1)
        phi_{y=0} - the model parameter that matches p(y=0)
        phi_{k|y=1} - the model parameter that matches p(x_j = k|y = 1) (for any j)
        phi_{k|y=0} - the model parameter that matches p(x_j = k|y = 0) (for any j)

    Refer to the remark from the assignment's pdf, about how to represent the parameters to avoid underflow.

    Args:
        matrix: A numpy array containing word counts for the training data
        labels: The binary (0 or 1) labels for that training data

    Returns: The trained model
    """

    model = dict.fromkeys(['phi_{y=0}', 'phi_{y=1}', 'phi_{k|y=0}', 'phi_{k|y=1}'])

    # *** START CODE HERE ***
```

$$\phi_j = \frac{1 + \sum_{i=1}^{n} 1\{z^{(i)} = j\}}{k + n}$$

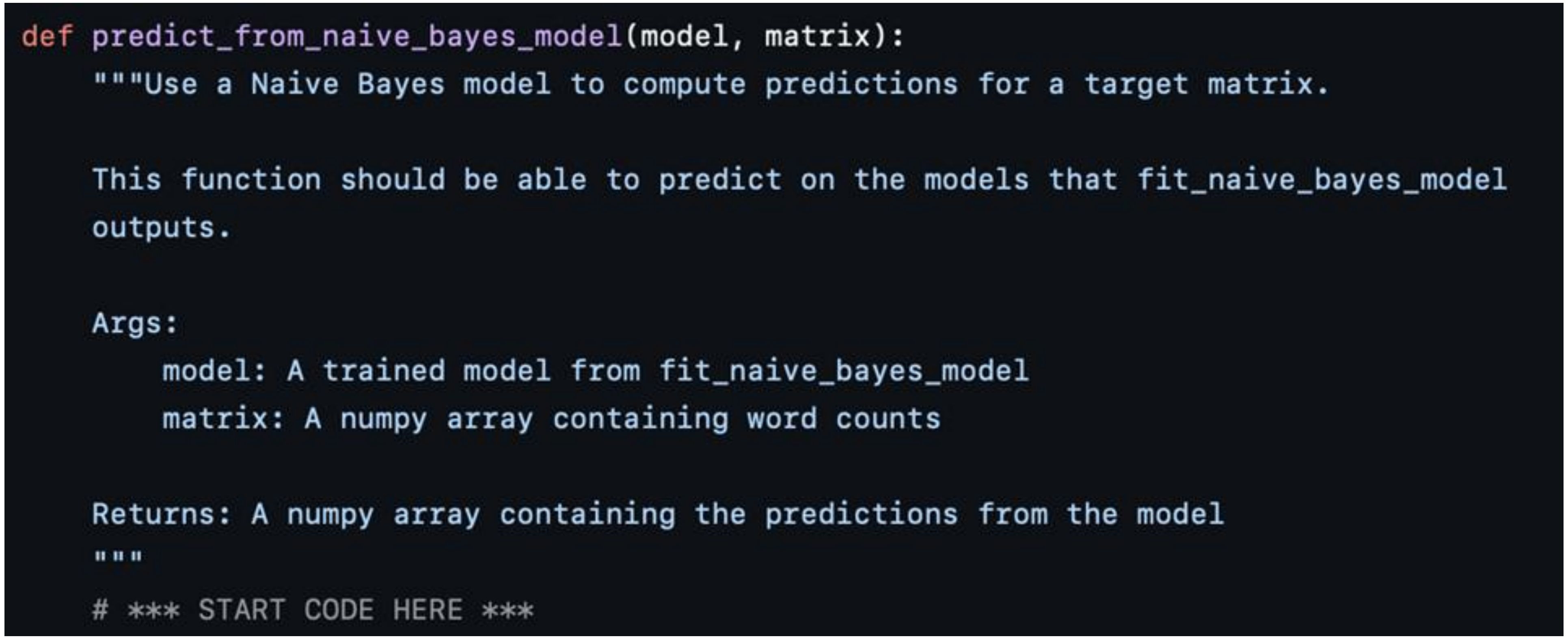
$$\phi_{j|y=1} = \frac{1 + \sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{2 + \sum_{i=1}^{n} 1\{y^{(i)} = 1\}}$$

$$\phi_{j|y=0} = \frac{1 + \sum_{i=1}^{n} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{2 + \sum_{i=1}^{n} 1\{y^{(i)} = 0\}}$$

our solution is 10 lines of code

Remark. If you implement Naive Bayes the straightforward way, you will find that the computed $p(x|y) = \prod_i p(x_i|y)$ often equals zero. This is because $p(x|y)$, which is the product of many numbers less than one, is a very small number. The standard computer representation of real numbers cannot handle numbers that are too small, and instead rounds them off to zero. (This is called "underflow.") You'll have to find a way to compute Naive Bayes' predicted class labels without explicitly representing very small numbers such as $p(x|y)$. [**Hint:** Think about using logarithms.]

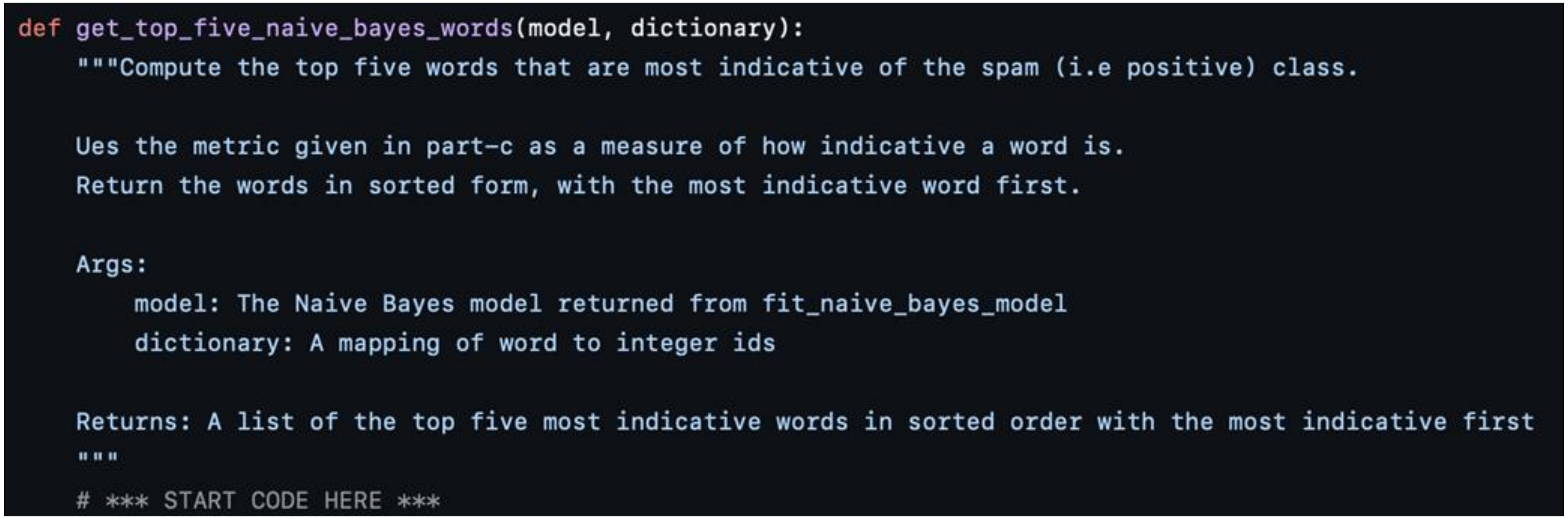
Take logs **before** putting the params in the model!



```python
def predict_from_naive_bayes_model(model, matrix):
    """Use a Naive Bayes model to compute predictions for a target matrix.

    This function should be able to predict on the models that fit_naive_bayes_model
    outputs.

    Args:
        model: A trained model from fit_naive_bayes_model
        matrix: A numpy array containing word counts

    Returns: A numpy array containing the predictions from the model
    """
    # *** START CODE HERE ***
```

our solution is 9 lines of code

$$P(y|x) = \log(P(y)) + \sum_{i=1}^{n} x_i \log(P(x^i|y)) = \boxed{\phi_y + \sum_{i=1}^{n} x_i \phi_{k|y}}$$



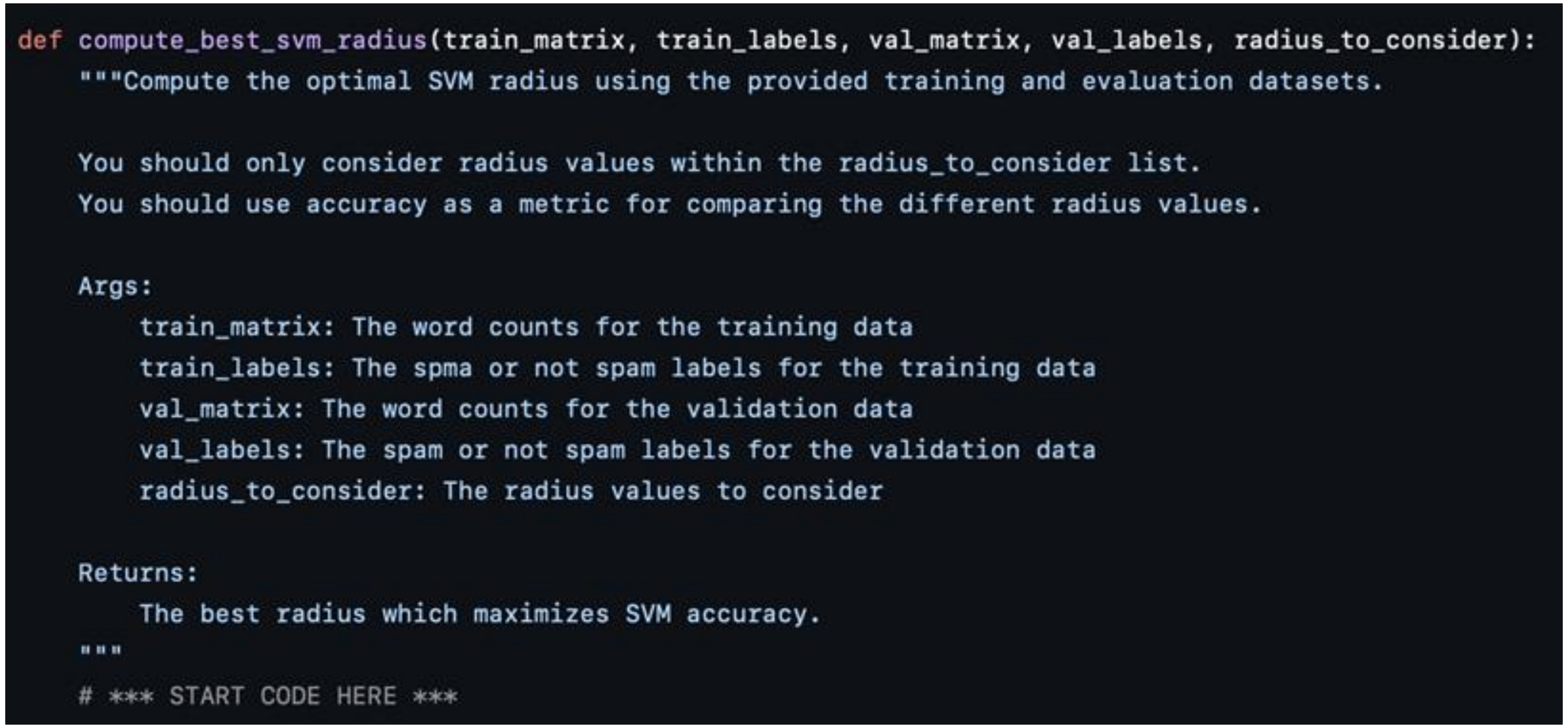
```
def get_top_five_naive_bayes_words(model, dictionary):
    """Compute the top five words that are most indicative of the spam (i.e positive) class.

    Ues the metric given in part-c as a measure of how indicative a word is.
    Return the words in sorted form, with the most indicative word first.

    Args:
        model: The Naive Bayes model returned from fit_naive_bayes_model
        dictionary: A mapping of word to integer ids

    Returns: A list of the top five most indicative words in sorted order with the most indicative first
    """
    # *** START CODE HERE ***
```

our solution is 3 lines of code

$$\log \frac{p(x_j = i|y=1)}{p(x_j = i|y=0)} = \log\left(\frac{P(\text{token } i|\text{email is SPAM})}{P(\text{token } i|\text{email is NOTSPAM})}\right) = \log(P(\text{token } i|\text{email is SPAM}) - \log(P(\text{token } i|\text{email is NOTSPAM}))$$

$$= \phi_{i|y=1} - \phi_{i|y=0}$$

The words that should come out are the most “spammy” words (i.e. “free”)



```
def compute_best_svm_radius(train_matrix, train_labels, val_matrix, val_labels, radius_to_consider):
    """Compute the optimal SVM radius using the provided training and evaluation datasets.

    You should only consider radius values within the radius_to_consider list.
    You should use accuracy as a metric for comparing the different radius values.

    Args:
        train_matrix: The word counts for the training data
        train_labels: The spma or not spam labels for the training data
        val_matrix: The word counts for the validation data
        val_labels: The spam or not spam labels for the validation data
        radius_to_consider: The radius values to consider

    Returns:
        The best radius which maximizes SVM accuracy.
    """
    # *** START CODE HERE ***
```

our solution is 9 lines of code

svm.py will be helpful

4. Double Descent on Linear Models

(a) **[3 points (Written)] Derive closed-form solution.**

In this question, we derive the closed-form solution of $\hat{\beta}_\lambda$. **Prove** that when $\lambda > 0$,

$$\hat{\beta}_\lambda = (X^\top X + \lambda I_{d\times d})^{-1} X^\top \vec{y} \tag{1}$$

(recall that $I_{d\times d} \in \mathbb{R}^{d\times d}$ is the identity matrix.)

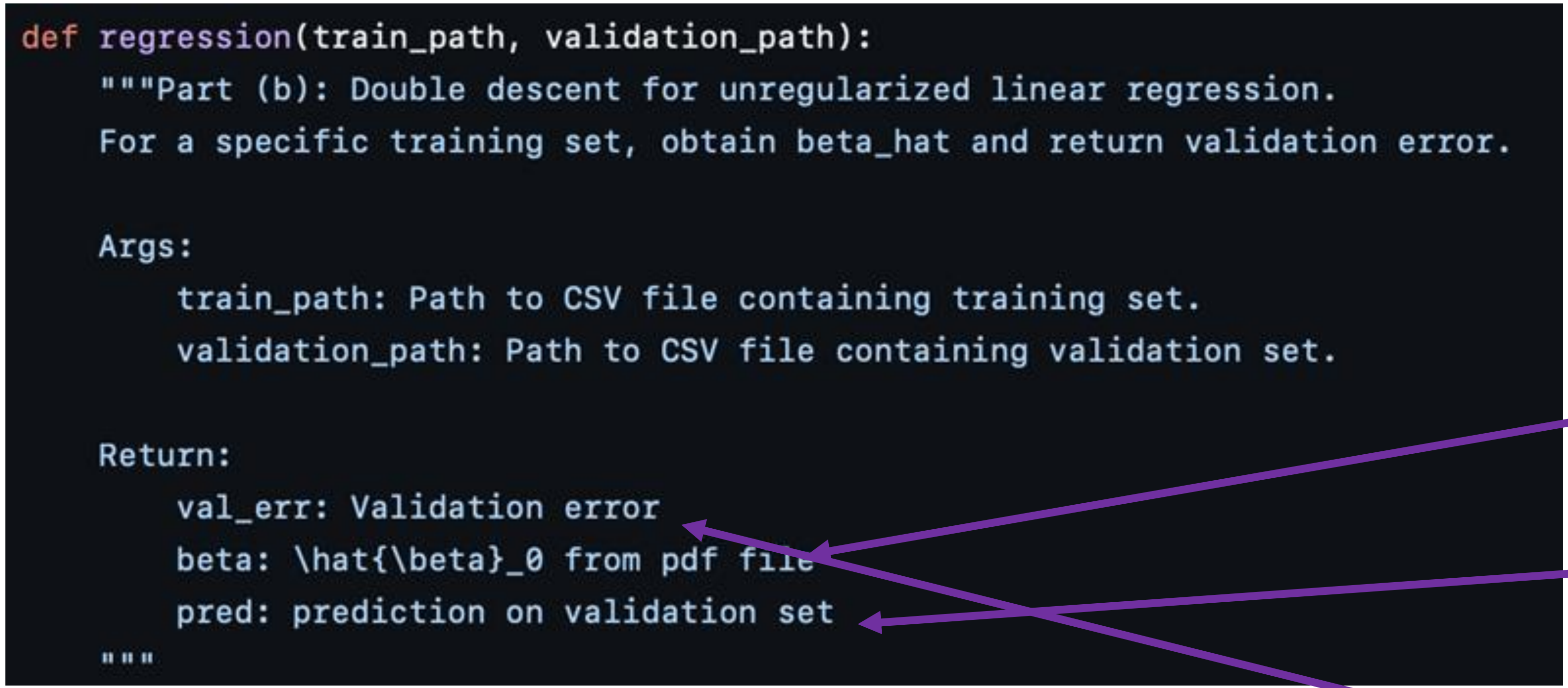
Note: $\lambda = 0$ is a special case here. When $\lambda = 0$, $(X^\top X + \lambda I_{d\times d})$ could be singular. Therefore, there might be more than one solutions that minimize $J_0(\beta)$. In this case, we define $\hat{\beta}_0$ in the following way:

$$\hat{\beta}_0 = (X^\top X)^+ X^\top \vec{y}. \tag{2}$$

where $(X^\top X)^+$ denotes the Moore-Penrose pseudo-inverse of $X^\top X$. You don't need to prove the case when $\lambda = 0$, but this definition is useful in the following sub-questions.

$$J_\lambda(\beta) = \frac{1}{2}\|X\beta - \vec{y}\|_2^2 + \frac{\lambda}{2}\|\beta\|_2^2$$

1. Take the gradient of J wrt. ß
2. Set it equal to 0 and solve



```
def regression(train_path, validation_path):
    """Part (b): Double descent for unregularized linear regression.
    For a specific training set, obtain beta_hat and return validation error.

    Args:
        train_path: Path to CSV file containing training set.
        validation_path: Path to CSV file containing validation set.

    Return:
        val_err: Validation error
        beta: \hat{\beta}_0 from pdf file
        pred: prediction on validation set
    """
```

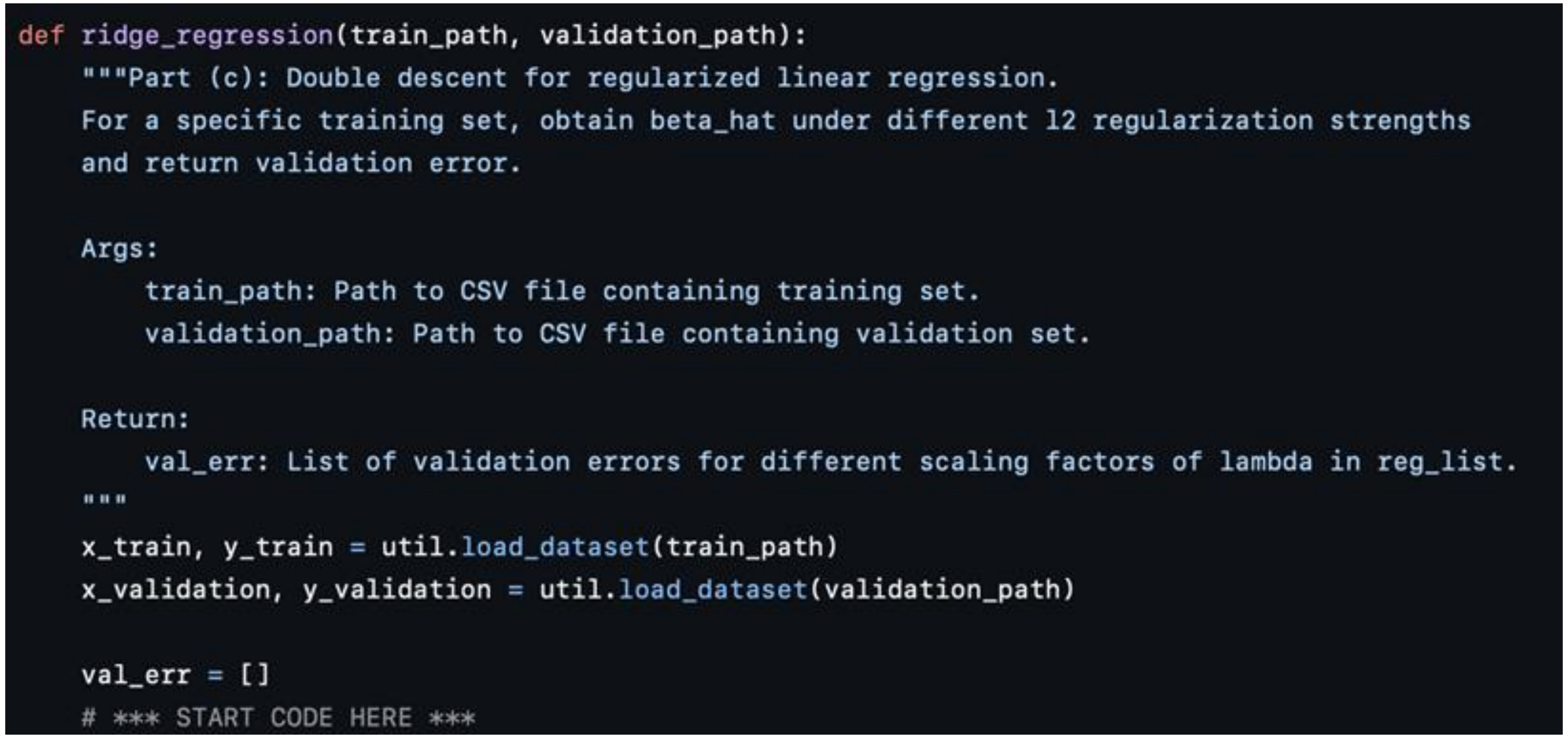
Do in this order:

$$\hat{\beta}_0 = (X^\top X)^+ X^\top \vec{y}.$$

$$X_v \hat{\beta}$$

$$\text{MSE}(\hat{\beta}) = \frac{1}{2m} \|X_v \hat{\beta} - \vec{y}_v\|_2^2.$$

our solution is 4 lines of code



```python
def ridge_regression(train_path, validation_path):
    """Part (c): Double descent for regularized linear regression.
    For a specific training set, obtain beta_hat under different l2 regularization strengths
    and return validation error.

    Args:
        train_path: Path to CSV file containing training set.
        validation_path: Path to CSV file containing validation set.

    Return:
        val_err: List of validation errors for different scaling factors of lambda in reg_list.
    """
    x_train, y_train = util.load_dataset(train_path)
    x_validation, y_validation = util.load_dataset(validation_path)

    val_err = []
    # *** START CODE HERE ***
```

our solution is 7 lines of code

1. Iteratate through reg_list, defined at the top of the file
2. Compute regularized ß $\hat{\beta}_\lambda = (X^\top X + \lambda I_{d\times d})^{-1} X^\top \vec{y}$
3. Use same equations for predictions and error
4. Append val_err

Good luck!