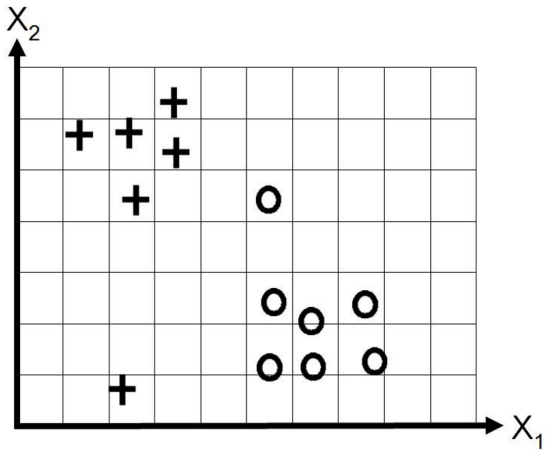
USC CSCI 467 Intro to Machine Learning	**Assignment 1** Due: September 23, 2024, 11:59pm	Fall 2024 Instructor: Jieyu Zhao

This assignment has 4 questions, for a total of 100 points. Make sure you also download the `hw1.zip` file from the course website.

When submitting on Gradescope, note that **you must make a submission both for the written portion and programming portion.** For the programming portion, **upload ONLY** `linreg.py` and `naivebayes.py` with your completed solution. (There is an "autograder" which will not actually grade your code but it will run it, and should return 0 if it encountered an error and 100 otherwise. Do not change the filenames of the files you upload.) **Please still include the output of your code in the PDF report when requested in the problems.**

Question 1: Linear Decision Boundaries (22 points)

(Adapted from PML Exercise 10.2) Consider the two-dimensional classification dataset below (`+`'s are positive examples, `o`'s are negative examples).



We will fit a logistic regression model with parameters $w \in \mathbb{R}^2$ and $b \in \mathbb{R}$ to this data, i.e., $p(y = 1 \mid x; w, b) = \sigma(w_1 x_1 + w_2 x_2 + b)$.

(a) (3 points) Suppose we fit a logistic regression model with no regularization. In other words, we are minimizing

$$L(w, b) = -\sum_{i=1}^{n} \log p(y = y^{(i)} \mid x^{(i)}; w, b)$$

Draw a possible decision boundary that would be learned. (You can do this in MS Paint or your favorite image editor; the image itself is included in `hw1.zip` so you can just copy and edit it.) How many classification errors does your model make on the training set? (i.e., how many examples are classified incorrectly?)

(b) (3 points) Suppose we apply strong regularization only to the w_1 parameter. That is, we minimize

$$L(w, b) = \left(-\sum_{i=1}^{n} \log p(y = y^{(i)} \mid x^{(i)}; w, b) \right) + \lambda w_1^2$$

where λ is a very large number, so that w_1 is forced to be 0. Draw a possible decision boundary that would result. Explain in 1-2 sentences how you chose this decision boundary. How many classification errors does your model make?

(c) (3 points) Now assume we only apply strong regularization to w_2, forcing w_2 to 0. Draw a possible decision boundary that would result. Explain in 1-2 sentences how you chose this decision boundary. How many classification errors does your model make?

(d) (3 points) Finally, assume we apply strong regularization only to b, forcing b to 0. Draw a possible decision boundary that would result. Explain in 1-2 sentences how you chose this decision boundary. How many classification errors does your model make?

(e) (4 points) Take any decision boundary defined by parameters w and b. Suppose we multiply w and b by 1000. What happens to the decision boundary? What happens to the margins of the correctly classified examples and the incorrectly classified examples? (Note that the margin on example (x, y) is defined as $y \cdot (w^\top x + b)$.) What does this do to the values of $p(y \mid x; w, b)$?

(f) (6 points) Now consider each of the four scenarios from parts a, b, c, and d. In each case, suppose we take the w and b that created the decision boundary you drew, and repeatedly multiply w and b by 1000. Will the logistic regression loss function (ignoring the regularization term) decrease or increase? Provide a separate answer for each of the four scenarios (a, b, c, and d), and explain your reasoning. (Hint: Recall that we can write the logistic loss as the function $g(z) = -\log \sigma(z)$ applied to the margin. Based on your answer to the previous part, look at what happens to the margins for every example in each scenario.)

Question 2: Regression with Laplacian Noise (23 points)

In class, we saw that linear regression "falls out" as the right algorithm to use if we assume that the data is generated by a linear function plus Gaussian noise. In this problem, we will see what changes when we assume that the noise is not Gaussian but instead follows a different distribution called the **Laplace** distribution.

The Laplace distribution is parameterized by a mean μ and a scale parameter b.[1] The probability density function is

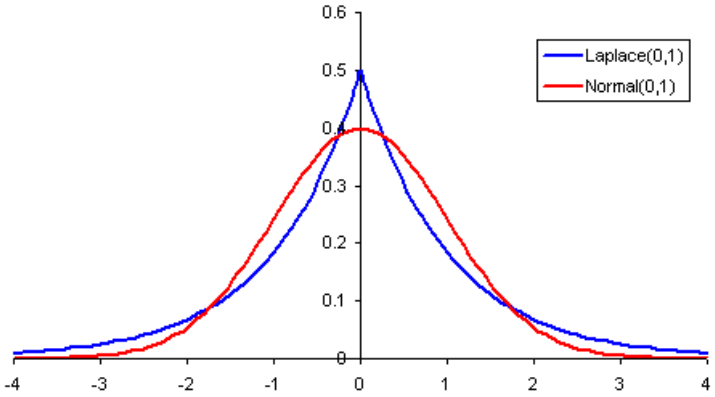
$$p_{\text{Laplace}}(x; \mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right).$$

Recall that the corresponding pdf for the Gaussian is

$$p_{\text{Gauss}}(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

We can visualize the difference between the Gaussian and Laplacian distributions by plotting their pdfs (with mean 0 and variance/scale parameter of 1) below:

[1] The letter b is commonly used to denote the scale parameter of the Laplace distribution. It is unrelated to the "bias" parameter b that we have used sometimes in linear and logistic regression.



The Laplacian distribution has a sharp peak at 0 instead of a smooth hump, and it has larger density than the Gaussian in the regions far away from 0 (sometimes referred to as "fatter tails").

Now let's use the Laplace distribution as our noise model for regression. We will assume that the examples are independently drawn from the distribution

$$p(y \mid x) = p_{\text{Laplace}}(y; w^\top x, b)$$

where b is some fixed constant. This says that y is a random variable centered around $w^\top x$ with noise distributed according to a Laplace distribution with scale parameter b. (In class, we made the exact same assumption but with a Gaussian distribution)

(a) (3 points) We are given a training dataset $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ of (x, y) pairs. Write down the equation for $\mathcal{L}(w)$, the likelihood of the the parameters w, assuming the Laplacian noise model.

(b) (3 points) By the principle of maximum likelihood estimation, doing machine learning here corresponds to finding w that maximizes $\mathcal{L}(w)$. Rewrite this as an optimization problem where you *minimize* a loss function with respect to w. Make this loss function an **average** over the training examples. Simplify your expression as much as possible (e.g., remove constants that are unnecessary to the optimization problem).

(c) (5 points) We want to learn w by gradient descent on this objective. Write the gradient update rule for w. Your answer should be a formula for $w^{(t)}$, the current guess for w, in terms of $w^{(t-1)}$, the previous guess for w. Use η to denote the learning rate.

(Note: Don't worry too much about what happens if $w^\top x = y$ exactly. You may use the "sign" function $\text{sign}(z)$, which is 1 if $z > 0$, -1 if $z < 0$, and 0 if $z = 0$, i.e. it returns the correct sign depending on whether z is positive or negative.)

(d) Let's compare this update rule with the one for linear regression. **For the first three sub-parts**, assume that we are doing one iteration of gradient update on the loss for a **single training point**, (x, y). And **for all sub-part**s, assume we use the same learning rate for all models.

i. (2 points) When the model's prediction, $w^\top x$, is pretty close to y, will Laplacian regression or standard linear regression make a larger update to w?

ii. (2 points) If $w^\top x$ is very far from y, will Laplacian regression or linear regression make a larger update to w?

iii. (2 points) If we initialize both models with the same $w^{(0)}$, will the gradient updates for Laplacian regression and linear regression always go in the same direction? Why or why not?

iv. (2 points) Still starting from the same $w^{(0)}$. But now suppose we do a gradient update on the **entire training dataset**, instead of just a single example. Will the gradient updates for Laplacian regression and linear regression always go in the same direction? Why or why not?

(e) (4 points) Suppose we are comparing two classifiers with weight vectors w and $\tilde{w}$. w makes small errors on every training example. $\tilde{w}$ predicts perfectly on most training examples, but has very high error on one training example. Your friend Gus decides to choose which of w and $\tilde{w}$ is better by measuring the standard linear regression loss (based on Gaussian noise), while your other friend Laura decides to compare them based on the Laplacian noise loss you have derived. Who is likely to favor w, and who is likely to favor $\tilde{w}$? Explain your reasoning.

Question 3: Linear Regression and Polynomial Features (23 points)

In this problem, we will implement linear regression and experiment see how different choices of features influence training error and test error.

Before you get started, take a look at `train.tsv`. Each row is an x and y value separated by a tab character. The `dev.tsv` and `test.tsv` files are structured similarly. The starter code already provides code that reads these files into a `numpy` array.

Setup: For this question and the next question, you will need to first do the following:

1. Download `hw1.zip` from the course website and extract the files.
2. Install the required packages:

   ```
   pip3 install -r requirements.txt
   ```

   You may first need to install pip, the python package installer.

(a) (5 points) Implement linear regression using gradient descent. You will have to modify the following functions:

- `predict(w, X)`, which takes in a parameter vector $w \in \mathbb{R}^d$ and matrix $X \in \mathbb{R}^{m \times d}$ for any integer m, and returns a vector $\hat{y} \in \mathbb{R}^m$ consisting of the model's prediction for each row of X. **For full credit, your solution should not use any for loops.**
- `train_gradient_descent(X_train, y_train)`, which takes in a training dataset consisting of a matrix $X \in \mathbb{R}^{n \times d}$ and vector $y \in \mathbb{R}^n$, where n is the number of training examples, d is the dimension of the feature vector for each example, and y_i is the label corresponding to the i-th row of X. The function returns a parameter vector $w \in \mathbb{R}^d$ chosen by gradient descent. **For full credit, your solution should only use a single for loop.** (Note: Yes, this is very similar to the demo from class.)

Now run the program to train on the training dataset and evaluate on the development set:

```
python3 linreg.py -a gradient_descent
```

This code will return the root mean squared error (RMSE) over the development set, i.e., the square root of the average across the development set D_{dev} of the squared difference

between our prediction and the true value:

$$\sqrt{\frac{1}{|D_{\text{dev}}|}\sum_{(x,y)\in D_{\text{dev}}}(w^\top x - y)^2}.$$

Note that this is the square root of the objective we use for training. In practice it can be more intuitive to look at the RMSE because it is in the same "units" as the y's (similar to the difference between standard deviation and variance).

You should get a RMSE of roughly 0.9454 on the development set. Once your code passes this sanity check, run on the test set by adding the `--test` flag to the python command. **Report the test set RMSE**.

(b) (6 points) Recall that for linear regression, we can also directly compute the closed form via the Normal Equations. Implement the function `train_normal_equations(X_train, y_train)`. Once you have done this, you can test your code by running:

```
python3 linreg.py -a normal
```

You should use the `numpy.linalg.pinv` function (it's already been imported in the starter code), which takes the pseudoinverse of a matrix. This should produce the same final output as your gradient descent code.

(c) (4 points) Now we will see what happens when we add more features. Look at the `featurize(x, d=1)` function. Currently, this function returns a list consisting of $[1, x]$. This corresponds to having a bias term (which you can think of as a 0-th degree polynomial) and a linear term. To increase the expressivity of our model, let's add monomial features corresponding to higher-degree polynomials. Change this function so that when provided $d > 1$, it returns the $d+1$-dimensional feature vector $[1, x, x^2, \ldots, x^d]$.

Now let's try running `linreg.py` again with different values of d. Run the following command:

```
python3 linreg.py -a normal -d 1:20
```

This will automatically try $d = 1$ through $d = 20$, printing the train and dev losses of each. and plot the train and dev RMSE's in a figure called `rmse_vs_degree.png`. Based on these results, what is the optimal degree to use? Let's call this degree d^*. **Report d^* and the test error when using d^*, and include the figure plotting RMSE vs. Degree.** (Note: You can also use `--degree [d]` to just run with a single degree instead of trying a range. Also, choose d^* based on the RMSE printout values for having a more accurate response instead of eyeballing the graph.)

(d) (4 points) `linreg.py` accepts another flag called `-n [n]` which reduces the number of training examples to n by just selecting the first n examples. This essentially allows us to simulate what would happen if we only had a smaller training dataset. Re-run the previous part with `-n 100` and report all the same results (note that the full training set has 1000 examples; also note that you may want to save the plot from the previous part somewhere else so it doesn't get overwritten). How do d^* and the final test error change? Explain why these changes make sense in 1-2 sentences. Again, choose d^* based on the RMSE printout values instead of manually examining the graph for having a more accurate response.

(e) (4 points) `linreg.py` accepts one more flag called `-p [d1,d2,d3]` that takes in a comma-separated list of degrees. It will plot the actual function that has been learned using each degree in the list, along with the development set data, and write this plot to `predictions_linreg.png`. Run the following command, replacing `d_star` with the d^* you found in the previous part:

```
python3 linreg.py -a normal -d 1:20 -n 100 -p 1,2,d_star,20
```

In your report, include the resulting figure and comment on the four curves. Does each curve appear to suffer from underfitting, overfitting, or neither? Explain your answers.

Question 4: Author Attribution with Naive Bayes (32 points)

In this question, you will build a Naive Bayes classifier to try to predict which famous author wrote a given piece of text.

Our dataset consists of sentences taken from books written by four authors:

1. Jane Austen: Sense and Sensibility, Pride and Prejudice, Emma
2. Agatha Christie: The Murder of Roger Ackroyd, The Mysterious Affair at Styles, Poirot Investigates
3. Herman Melville: Moby Dick, Typee, Omoo
4. William Shakespeare: Romeo and Juliet, Othello, A Midsummer Night's Dream

The files `train.tsv`, `dev.tsv`, and `test.tsv` all contain examples sampled from the 12 books listed above; the examples were randomly partitioned into the train/dev/test splits. Each row of each file contains an author ("austen", "christie", "melville", or "shakespeare"), the book (just for reference; we won't use this in the code), and a passage that has been lower-cased and split into space-separated words. Some minor preprocessing has been done, e.g., punctuation marks are considered their own words in this format.

We will use the Multinomial event model with Naive Bayes. Recall from lecture that in the multinomial event model, we imagine a generative process in which a document consisting of words $x_1, \ldots, x_m$ is generated one word at a time. Each word is sampled independently from a multinomial distribution over the vocabulary of possible words (this is the Naive Bayes assumption). We further assume that the word distribution for each *position* in the document is the same, i.e., $p(x_j \mid y)$ is the same for all indices j.

(a) (15 points) Implement the Multinomial Naive Bayes model. This involves implementing the following functions:

- `get_label_counts(train_data)`, which counts the number of examples that occur with each label in the dataset.
- `get_word_counts(train_data)`, which counts the number of times each (word, label) pair occurs in the dataset.
- `predict(words, label_counts, word_counts, vocabulary)`, which makes a prediction given an input consisting of `words`. Use Laplace Smoothing with $\lambda = 1$ when making predictions. You only need to apply Laplace smoothing to your calculations of $P(x_j \mid y)$, not to the prior distribution over $P(y)$. **Note: To prevent numerical underflow, you will want to work in log space.**

Once you have implemented these, run the following:

`python3 naivebayes.py -e dev`

to evaluate on the development set. You should get an accuracy of 96.493%. Once you pass this sanity check, evaluate on the test set using `-e test` and **report the test accuracy**. (Note: Our reference implementation takes about 15 seconds total to run on a three-year-old laptop. If your implementation is considerably slower, consider if you can precompute any quantities to make things run faster.)

(b) (2 points) Our classifier seems to work pretty well! Now let's give it a harder test. Run the same command but with `-e newbooks`. This will test on the data in `newbooks.tsv`, which contains examples sampled from unseen *books* written by the same authors:

1. Jane Austen: Persuasion
2. Agatha Christie: The Murder on the Links
3. Herman Melville: Pierre; or, The Ambiguities
4. William Shakespeare: Hamlet

What is the accuracy on this new dataset?

(c) (9 points) Our model's accuracy dropped by a lot! To understand this, let's try to understand what our model has actually learned. In particular, one reasonable strategy is to try to understand which features the model relies on most to make its predictions. In our case, that means trying to understand which words are most strongly associated with a particular label.

To do this, let's imagine taking every individual word w in our dataset, and feeding it as the input to the Naive Bayes model. (Essentially, we're feeding in a "document" consisting of a single word.) For any label $\tilde{y}$, the Naive Bayes model can compute the posterior probability $P(y = \tilde{y} \mid w)$. Recall that the formula for this is:

$$P(y = \tilde{y} \mid w) = \frac{P(\tilde{y}) \cdot P(w \mid \tilde{y})}{\sum_{\hat{y} \in A} P(\hat{y}) \cdot P(w \mid \hat{y})}$$

where A is the set of authors (i.e., the set of possible labels). If this probability is large, that is a sign that word w is strongly associated with label $\tilde{y}$.

Write code in `analyze_counts(word_counts, vocabulary)` to find the top 10 words associated with each label, i.e., the ten w with the largest value of $P(y = \tilde{y} \mid w)$ for each label $\tilde{y}$. Print them out and report the list of words paired with the probability of the associated label. Use the `-a` flag to cause the code to run `analyze_counts`. What patterns do you see? Comment on at least two different types of word-label associations. Finally, comment on why your findings could explain the worse performance on new books.

(d) (3 points) We can also use this analysis to understand the effect of Laplace Smoothing. Try repeating the previous part but without Laplace Smoothing. How do the values of $P(y = \tilde{y} \mid w)$ change? What types of words now appear at the top? Why does that happen?

(e) (3 points) For the last step in our investigation, let's break down the errors per book. You probably have noticed that after evaluating on the dev, test, or newbooks datasets, the program spits out what's known as a *confusion matrix*. Each row shows the examples corresponding to one true label, and each column shows how many times the model predicted each label on that subset of the examples.

Based on the confusion matrix for the newbooks data, which of the four new books was the hardest to classify? Why do you think this book was difficult for the model? (Hint: It may help you to read up on what all the books are about.)