XCS229 Problem Set 5

Due Sunday, June 11 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs229-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

1. **[3.50 points (Written)] PCA**

Note for this problem set you can use XCS229's PCA notes as supplemental material to compliment your approach since the course does not explicitly have lecture videos around PCA.

Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(n)}\}$. Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector u, let $f_u(x)$ be the projection of point x onto the direction given by u. I.e., if $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$, then

$$f_u(x) = \arg\min_{v \in \mathcal{V}} || \, x - v \, ||^2 \, .$$

Show that the unit-length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

$$\arg \min_{u: u^T u = 1} \sum_{i=1}^{n} | \, x^{(i)} - f_u(x^{(i)}) \, |_2^2 \quad .$$

gives the first principal component.

Remark. If we are asked to find a k-dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the k-dimensional subspace spanned by the first k principal components of the data. This problem shows that this result holds for the case of $k = 1$.

2. **Semi-supervised EM**

Expectation Maximization (EM) is a classical algorithm for unsupervised learning (*i.e.,* learning with hidden or latent variables). In this problem we will explore one of the ways in which the EM algorithm can be adapted to the semi-supervised setting, where we have some labeled examples along with unlabeled examples.

In the standard unsupervised setting, we have $n \in \mathbb{N}$ unlabeled examples $\{x^{(1)}, \ldots, x^{(n)}\}$. We wish to learn the parameters of $p(x, z; \theta)$ from the data, but $z^{(i)}$'s are not observed. The classical EM algorithm is designed for this very purpose, where we maximize the intractable $p(x; \theta)$ indirectly by iteratively performing the E-step and M-step, each time maximizing a tractable lower bound of $p(x; \theta)$. Our objective can be concretely written as:

$$\begin{aligned}\ell_{\text{unsup}}(\theta) &= \sum_{i=1}^{n} \log p(x^{(i)}; \theta) \\ &= \sum_{i=1}^{n} \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)\end{aligned}$$

Now, we will attempt to construct an extension of EM to the semi-supervised setting. Let us suppose we have an *additional* $\tilde{n} \in \mathbb{N}$ labeled examples $\{(\tilde{x}^{(1)}, \tilde{z}^{(1)}), \ldots, (\tilde{x}^{(\tilde{n})}, \tilde{z}^{(\tilde{n})})\}$ where both x and z are observed. We want to simultaneously maximize the marginal likelihood of the parameters using the unlabeled examples, and full likelihood of the parameters using the labeled examples, by optimizing their weighted sum (with some hyperparameter α). More concretely, our semi-supervised objective $\ell_{\text{semi-sup}}(\theta)$ can be written as:

$$\begin{aligned}\ell_{\text{sup}}(\theta) &= \sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta) \\ \ell_{\text{semi-sup}}(\theta) &= \ell_{\text{unsup}}(\theta) + \alpha \ell_{\text{sup}}(\theta)\end{aligned}$$

We can derive the EM steps for the semi-supervised setting using the same approach and steps as before. You are *strongly encouraged* to show to yourself (no need to include in the write-up) that we end up with:

E-step (semi-supervised)

For each $i \in \{1, \ldots, n\}$, set

$$Q_i^{(t)}(z^{(i)}) := p(z^{(i)} | x^{(i)}; \theta^{(t)})$$

M-step (semi-supervised)

$$\theta^{(t+1)} := \arg\max_{\theta} \left[\sum_{i=1}^{n} \left(\sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i^{(t)}(z^{(i)})} \right) + \alpha \left(\sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta) \right) \right]$$

(a) **[2.50 points (Written)]**

Convergence. First we will show that this algorithm eventually converges. In order to prove this, it is sufficient to show that our semi-supervised objective $\ell_{\text{semi-sup}}(\theta)$ monotonically increases with each iteration of E and M step. Specifically, let $\theta^{(t)}$ be the parameters obtained at the end of t EM-steps. Show that $\ell_{\text{semi-sup}}(\theta^{(t+1)}) \geq \ell_{\text{semi-sup}}(\theta^{(t)})$.

Semi-supervised GMM

Now we will revisit the Gaussian Mixture Model (GMM), to apply our semi-supervised EM algorithm. Let us consider a scenario where data is generated from $k \in \mathbb{N}$ Gaussian distributions, with unknown means $\mu_j \in \mathbb{R}^d$ and covariances $\Sigma_j \in \mathbb{S}_+^d$ where $j \in \{1, \ldots, k\}$. We have n data points $x^{(i)} \in \mathbb{R}^d, i \in \{1, \ldots, n\}$, and each data point has a corresponding latent (hidden/unknown) variable $z^{(i)} \in \{1, \ldots, k\}$ indicating which distribution $x^{(i)}$ belongs

to. Specifically, $z^{(i)} \sim \text{Multinomial}(\phi)$, such that $\sum_{j=1}^{k} \phi_j = 1$ and $\phi_j \geq 0$ for all j, and $x^{(i)}|z^{(i)} \sim \mathcal{N}(\mu_{z^{(i)}}, \Sigma_{z^{(i)}})$ i.i.d. So, μ, Σ, and ϕ are the model parameters.

We also have additional $\tilde{n}$ data points $\tilde{x}^{(i)} \in \mathbb{R}^d, i \in \{1, \ldots, \tilde{n}\}$, and an associated *observed* variable $\tilde{z}^{(i)} \in \{1, \ldots, k\}$ indicating the distribution $\tilde{x}^{(i)}$ belongs to. Note that $\tilde{z}^{(i)}$ are known constants (in contrast to $z^{(i)}$ which are unknown *random* variables). As before, we assume $\tilde{x}^{(i)}|\tilde{z}^{(i)} \sim \mathcal{N}(\mu_{\tilde{z}^{(i)}}, \Sigma_{\tilde{z}^{(i)}})$ i.i.d.

In summary we have $n + \tilde{n}$ examples, of which n are unlabeled data points x's with unobserved z's, and $\tilde{n}$ are labeled data points $\tilde{x}^{(i)}$ with corresponding observed labels $\tilde{z}^{(i)}$. The traditional EM algorithm is designed to take only the n unlabeled examples as input, and learn the model parameters μ, Σ, and ϕ.

Our task now will be to apply the semi-supervised EM algorithm to GMMs in order to also leverage the additional $\tilde{n}$ labeled examples, and come up with semi-supervised E-step and M-step update rules specific to GMMs. Whenever required, you can cite the lecture notes for derivations and steps.

(b) **[2.50 points (Written)] Semi-supervised E-Step.** Clearly state which are all the latent variables that need to be re-estimated in the E-step. Derive the E-step to re-estimate all the stated latent variables. Your final E-step expression must only involve x, z, μ, Σ, ϕ and universal constants.

(c) **[5.50 points (Written)] Semi-supervised M-Step.** Clearly state which are all the parameters that need to be re-estimated in the M-step. Derive the M-step to re-estimate all the stated parameters. Specifically, derive closed form expressions for the parameter update rules for $\mu^{(t+1)}$, $\Sigma^{(t+1)}$ and $\phi^{(t+1)}$ based on the semi-supervised objective.

(d) **[6 points (Coding)] Classical (Unsupervised) EM Implementation.** For this sub-question, we are only going to consider the n unlabeled examples. Follow the instructions in `src-semi_supervised_em/submission.py` to implement the traditional EM algorithm, and run it on the unlabeled data-set until convergence. More specifically, please complete the `main` and `run_em` functions. Note: feel free to create your own helper functions in the development of your solution.

Autograder test case `2d-1-basic` can be used to verify a correct implementation. Before running the test case, change line 92 to `skip = False` (this test is skipped by default to make the autograder faster). It will run three trials and use the provided plotting function to construct a scatter plot of the resulting assignments to clusters (one plot for each trial). The output plot will indicate cluster assignments by assigning unique colors for each cluster (*i.e.,* the cluster which had the highest probability in the final E-step). Do not submit these plots; they are not graded.

Your plots should look similar to the following:

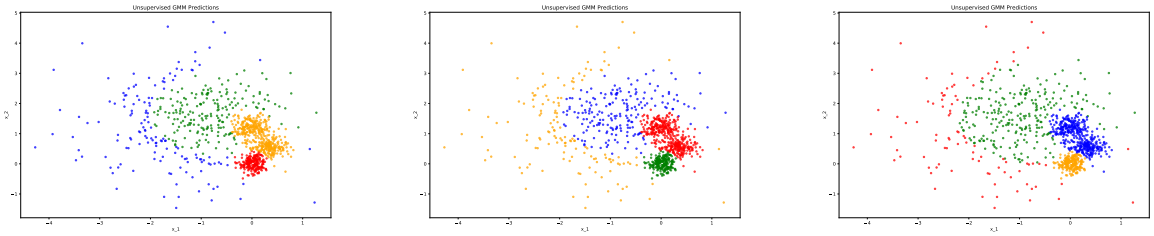


Figure 1: Predictions made by GMM model with unsupervised EM.

(e) **[7 points (Coding)] Semi-supervised EM Implementation.** Now we will consider both the labeled and unlabeled examples (a total of $n + \tilde{n}$), with 5 labeled examples per cluster. We have provided starter code for splitting the dataset into matrices `x` and `x_tilde` of unlabeled and labeled examples respectively. Add to your code in `src-semi_supervised_em/submission.py` to implement the modified EM algorithm, and run it on the dataset until convergence. More specifically, you will complete the `main` and `run_semi_supervised_em` functions. Note: feel free to create your own helper functions in the development of your solution.

Autograder test case `2e-1-basic` can be used to verify a correct implementation. Before running the test case,

change line 143 to `skip = False` (this test is skipped by default to make the autograder faster). It will runcreate a plot for each trial, as done in the previous sub-question.

Your plots should look similar to the following (your plots are not graded):

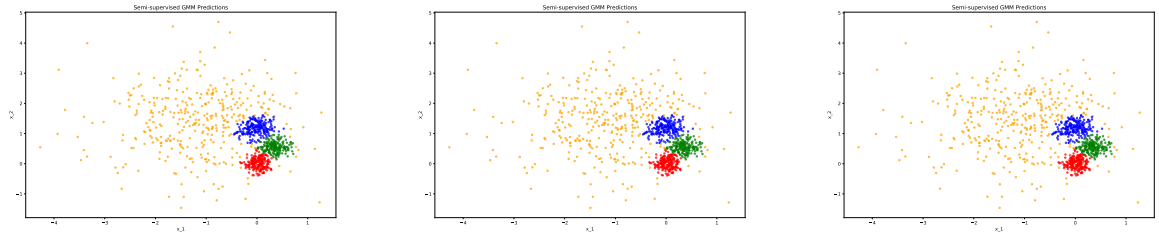


Figure 2: Predictions made by GMM model with semi-supervised EM.

(f) **[2.50 points (Written)] Comparison of Unsupervised and Semi-supervised EM.** Briefly describe the differences you saw in unsupervised *vs.* semi-supervised EM for each of the following:

i. Number of iterations taken to converge.

ii. Stability (*i.e.,* how much did assignments change with different random initializations?)

iii. Overall quality of assignments.

Note: The dataset was sampled from a mixture of three low-variance Gaussian distributions, and a fourth, high-variance Gaussian distribution. This should be useful in determining the overall quality of the assignments that were found by the two algorithms.

3. **Independent components analysis**

While studying Independent Component Analysis (ICA) in class, we made an informal argument about why Gaussian distributed sources will not work. We also mentioned that any other distribution (except Gaussian) for the sources will work for ICA, and hence used the logistic distribution instead. In this problem, we will go deeper into understanding why Gaussian distributed sources are a problem. We will also derive ICA with the Laplace distribution, and apply it to the cocktail party problem.

Reintroducing notation, let $s \in \mathbb{R}^d$ be source data that is generated from d independent sources. Let $x \in \mathbb{R}^d$ be observed data such that $x = As$, where $A \in \mathbb{R}^{d \times d}$ is called the *mixing matrix*. We assume A is invertible, and $W = A^{-1}$ is called the *unmixing matrix*. So, $s = Wx$. The goal of ICA is to estimate W. Similar to the notes, we denote w_j^T to be the j^{th} row of W. Note that this implies that the j^{th} source can be reconstructed with w_j and x, since $s_j = w_j^T x$. We are given a training set $\{x^{(1)}, \ldots, x^{(n)}\}$ for the following sub-questions. Let us denote the entire training set by the design matrix $X \in \mathbb{R}^{n \times d}$ where each example corresponds to a row in the matrix.

(a) **[2 points (Written)] Gaussian source**

For this sub-question, we assume sources are distributed according to a standard normal distribution, i.e $s_j \sim \mathcal{N}(0, 1), j = \{1, \ldots, d\}$. The log-likelihood of our unmixing matrix, as described in the notes, is

$$\ell(W) = \sum_{i=1}^{n} \left(\log \mid W \mid + \sum_{j=1}^{d} \log g'(w_j^T x^{(i)}) \right),$$

where g is the cumulative distribution function, and g' is the probability density function of the source distribution (in this sub-question it is a standard normal distribution). Whereas in the notes we derive an update rule to train W iteratively, for the cause of Gaussian distributed sources, we can analytically reason about the resulting W.

Try to derive a closed form expression for W in terms of X when g is the standard normal CDF. Deduce the relation between W and X in the simplest terms, and highlight the ambiguity (in terms of rotational invariance) in computing W.

(b) **[2.50 points (Written)] Laplace source.**

For this sub-question, we assume sources are distributed according to a standard Laplace distribution, i.e $s_i \sim \mathcal{L}(0, 1)$. The Laplace distribution $\mathcal{L}(0, 1)$ has PDF $f_{\mathcal{L}}(s) = \frac{1}{2} \exp(- \mid s \mid)$. With this assumption, derive the update rule for a single example in the form

$$W := W + \alpha(\ldots).$$

(c) **[4 points (Coding)] Cocktail Party Problem**

For this question you will implement the Bell and Sejnowski ICA algorithm, but assuming a Laplace source (as derived in part-b), instead of the Logistic distribution covered in class. The file `src-ica/mix.dat` contains the input data which consists of a matrix with 5 columns, with each column corresponding to one of the mixed signals x_i. The code for this question can be found in `src-ica/submission.py`.

Implement the `update_W` and `unmix` functions in `src-ica/submission.py`.

You can then run `python3 src-ica/submission.py` in order to split the mixed audio into its components. The mixed audio tracks are written to `mixed_i.wav` in the output folder. The split audio tracks are written to `split_i.wav` in the output folder.

To make sure your code is correct, you should listen to the resulting unmixed sources. (Some overlap or noise in the sources may be present, but the different sources should be pretty clearly separated.)

Submit the full unmixing matrix W (5×5) that you obtained, by including the W.txt the code outputs along with your code.

If you implemention is correct, your output `split_0.wav` should sound similar to the file `correct_split_0.wav` included with the source code. If you are on a Mac and if your default audio player is iTunes it does not update the audio file if you play the second time. So you will keep hearing the same old file unless you delete it from the playlist. As a workaround, view the audio files using Quicktime or VLC instead.

Note: In our implementation, we **anneal** the learning rate α (slowly decreased it over time) to speed up learning. In addition to using the variable learning rate to speed up convergence, one thing that we also do is choose a random permutation of the training data, and running stochastic gradient ascent visiting the training data in that order (each of the specified learning rates was then used for one full pass through the data).

Files to Submit: `src-ica/W.txt`, `src-ica/submission.py`

4. **Implicit Regularization**

Recall that in the overparameterized regime (where the number of parameters is larger than the number of samples), typically there are infinitely many solutions that can fit the training dataset perfectly, and many of them cannot generalize well (that is, they have large validation errors). However, in many cases, the particular optimizer we use (e.g., GD, SGD with particular learning rates, batch sizes, noise, etc.) tends to find solutions that generalize well. This phenomenon is called implicit regularization effect (also known as algorithmic regularization or implicit bias).

In this problem, we will look at the implicit regularization effect on two toy examples in the overparameterized regime: linear regression and a quadratically parameterized model. For linear regression, we will show that gradient descent with zero initialization will always find the minimum norm solution (instead of an arbitrary solution that fits the training data), and in practice, the minimum norm solution tends to generalize well. For a quadratically parameterized model, we will show that initialization and batch size also affect generalization.

(a) **[1 point (Written)]** Suppose we have a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, n\}$ where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for all $1 \leq i \leq n$. We assume the dataset is generated by a linear model without noise. That is, there is a vector $\beta^\star \in \mathbb{R}^d$ such that $y^{(i)} = (\beta^\star)^\top x^{(i)}$ for all $1 \leq i \leq n$. Let $X \in \mathbb{R}^{n \times d}$ be the matrix representing the inputs (i.e., the i-th row of X corresponds to $x^{(i)}$)) and $\vec{y} \in \mathbb{R}^n$ the vector representing the labels (i.e., the i-th row of $\vec{y}$ corresponds to $y^{(i)}$)):

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Then in matrix form, we can write $\vec{y} = X\beta^\star$. We assume that the number of examples is less than the number of parameters (that is, $n < d$).

We use the least-squares cost function to train a linear model:

$$J(\beta) = \frac{1}{2n}\|X\beta - \vec{y}\|_2^2. \tag{1}$$

In this sub-question, we characterize the family of global minimizers to Eq. (1). We assume that $XX^\top \in \mathbb{R}^{n \times n}$ is an invertible matrix. **Prove that** β achieves zero cost in Eq. (1) if and only if

$$\beta = X^\top (XX^\top)^{-1}\vec{y} + \zeta \tag{2}$$

for some ζ in the subspace orthogonal to all the data (that is, for some ζ such that $\zeta^\top x^{(i)} = 0, \forall 1 \leq i \leq n$.)

Note that this implies that there is an infinite number of β's such that Eq. (1) is minimized. We also note that $X^\top (XX^\top)^{-1}$ is the pseudo-inverse of X, but you don't necessarily need this fact for the proof.

(b) **[1 point (Written)]** We still work with the setup of part (a). Among the infinitely many optimal solutions of Eq. (1), we consider the *minimum norm* solution. Let $\rho = X^\top (XX^\top)^{-1}\vec{y}$. In the setting of (a), **prove that** for any β such that $J(\beta) = 0$, $\|\rho\|_2 \leq \|\beta\|_2$. In other words, ρ is the minimum norm solution.

Hint: As a intermediate step, you can prove that for any β in the form of Eq. (2),

$$\|\beta\|_2^2 = \|\rho\|_2^2 + \|\zeta\|_2^2.$$

(c) **[4 points (Coding)] Minimum norm solution generalizes well**

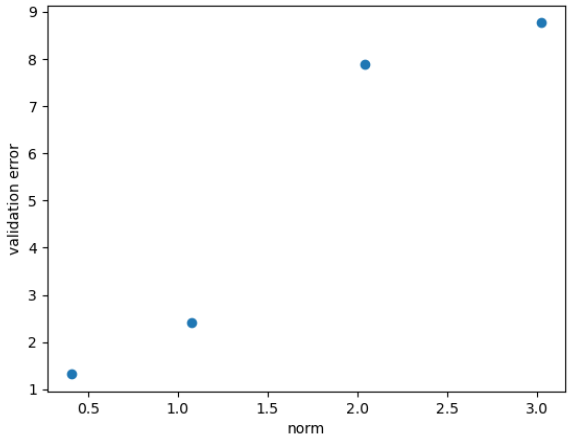
For this sub-question, we still work with the setup of parts (a) and (b). We use the following datasets:

`src-implicitreg/ir1_train.csv, ir1_valid.csv`

Each file contains $d + 1$ columns. The first d columns in the i-th row represents $x^{(i)}$, and the last column represents $y^{(i)}$. In this sub-question, we use $d = 200$ and $n = 40$.

Using the formula in sub-question (b), **compute** the minimum norm solution using the training dataset. Then, **generate** three other different solutions with zero costs and different norms using the formula in sub-question (a). The starter code is in `src-implicitreg/submission.py` and you should implement the

`get_minimum_norm_solution` and `get_different_n_solutions` functions. Your generated plot should demonstrate that the minimum norm solution generalizes well and should be similar to the following plot:



(d) **[2 points (Written)]** For this sub-question, we work with the setup of part (a) and (b). In this sub-question, you will prove that the gradient descent algorithm with *zero initialization* always converges to the minimum norm solution. Let $\beta^{(t)}$ be the parameters found by the GD algorithm at time step t. Recall that at step t, the gradient descent algorithm update the parameters in the following way

$$\beta^{(t)} = \beta^{(t-1)} - \eta \nabla J(\beta^{(t-1)}) = \beta^{(t-1)} - \frac{\eta}{n} X^\top (X\beta^{(t-1)} - \vec{y}). \tag{3}$$

As in sub-question (a), we also assume $XX^\top$ is an invertible matrix. **Prove** that if the GD algorithm with zero initialization converges to a solution $\hat{\beta}$ satisfying $J(\hat{\beta}) = 0$, then $\hat{\beta} = X^\top (XX^\top)^{-1}\vec{y} = \rho,$, that is, $\hat{\beta}$ is the minimum norm solution.

Hint: As a first step, you can prove by induction that if we start with zero initialization, $\beta^{(t)}$ will always be a linear combination of $\{x^{(1)}, x^{(2)}, \cdots, x^{(n)}\}$ for any $t \geq 0$. Then, for any $t \geq 0$, you can write $\beta^{(t)} = X^\top v^{(t)}$ for some $v^{(t)} \in \mathbb{R}^n$. As a second step, you can prove that if $\hat{\beta} = X^\top v^{(t)}$ for some $v^{(t)}$ and $J(\hat{\beta}) = 0$, then we have $\hat{\beta} = \rho$.

You don't necessarily have to follow the steps in this hint. But if you use the hint, you need to prove the statements in the hint.

(e) **[1 point (Written)]** In the following sub-questions, we consider a slightly more complicated model called quadratically parameterized model. A quadratically parameterized model has two sets of parameters $\theta, \phi \in \mathbb{R}^d$. Given a d-dimensional input $x \in \mathbb{R}^d$, the output of the model is

$$f_{\theta,\phi}(x) = \sum_{k=1}^{d} \theta_k^2 x_k - \sum_{k=1}^{d} \phi_k^2 x_k. \tag{4}$$

Note that $f_{\theta,\phi}(x)$ is linear in its input x, but non-linear in its parameters θ, ϕ. Thus, if the goal was to learn the function, one should simply just re-parameterize it with a linear model and use linear regression. However, here we insist on using the parameterization above in Eq. (4) in order to study the implicit regularization effect in models that are nonlinear in the parameters.

Notations: To simplify the equations, we define the following notations. For a vector $v \in \mathbb{R}^d$, let $v^{\odot 2}$ be its element-wise square (that is, $v^{\odot 2}$ is the vector $[v_1^2, v_2^2, \cdots, v_d^2] \in \mathbb{R}^d$.) For two vectors $v, w \in \mathbb{R}^d$, let $v \odot w$ be their element-wise product (that is, $v \odot w$ is the vector $[v_1 w_1, v_2 w_2, \cdots, v_d w_d] \in \mathbb{R}^d$.) Then our model can be written as

$$f_{\theta,\phi}(x) = x^\top (\theta^{\odot 2} - \phi^{\odot 2}). \tag{5}$$

Suppose we have a dataset $\{(x^{(i)}, y^{(i)}); i = 1, \cdots, n\}$ where $x^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for all $1 \le i \le n$, and

$$y^{(i)} = (x^{(i)})^\top ((\theta^\star)^{\odot 2} - (\phi^\star)^{\odot 2})$$

for some $\theta^\star, \phi^\star \in \mathbb{R}^d$. Similarly, we use $X \in \mathbb{R}^{n \times d}$ and $\vec{y} \in \mathbb{R}^n$ to denote the matrix/vector representing the inputs/labels respectively:

$$X = \begin{bmatrix} - & x^{(1)} & - \\ - & x^{(2)} & - \\ \vdots & \vdots & \vdots \\ - & x^{(n)} & - \end{bmatrix}, \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Let $J(\theta, \phi) = \frac{1}{4n} \sum_{i=1}^n (f_{\theta,\phi}(x^{(i)}) - y^{(i)})^2$ be the cost function.

First, when $n < d$ and $XX^\top$ is invertible, **prove** that there exists infinitely many optimal solutions with zero cost.

Hint: Find a mapping between the parameter β in linear model and the parameter θ, ϕ in quadratically parameterized model. Then use the conclusion in sub-question (a).

(f) **[4.50 points (Coding)] Implicit regularization of initialization**

We still work with the setup in part (e). For this sub-question, we use the following datasets:

`src-implicitreg/ir2_train.csv, ir2_valid.csv`

Each file contains $d + 1$ columns. The first d columns in the i-th row represents $x^{(i)}$, and the last column represents $y^{(i)}$. In this sub-question, we use $d = 200$ and $n = 40$.

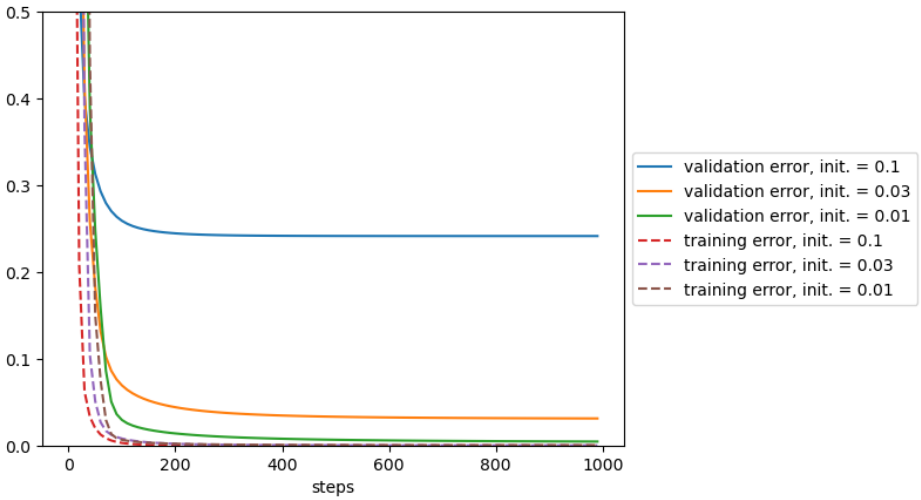
First of all, the gradient of the loss has the following form:

$$\nabla_\theta J(\theta, \phi) = \frac{1}{n} \sum_{i=1}^n ((x^{(i)})^\top (\theta^{\odot 2} - \phi^{\odot 2}) - y^{(i)})(\theta \odot x^{(i)}), \tag{6}$$

$$\nabla_\phi J(\theta, \phi) = -\frac{1}{n} \sum_{i=1}^n ((x^{(i)})^\top (\theta^{\odot 2} - \phi^{\odot 2}) - y^{(i)})(\phi \odot x^{(i)}). \tag{7}$$

You don't need to prove these two equations. They can be verified directly using the chain rule.

Using the formula above, run gradient descent with initialization $\theta = \alpha \mathbf{1}, \phi = \alpha \mathbf{1}$ with $\alpha \in \{0.1, 0.03, 0.01\}$ (where $\mathbf{1} = [1, 1, \cdots, 1] \in \mathbb{R}^d$ is the all-1's vector) and learning rate 0.08. We provide the starter code in `src-implicitreg/submission.py` and you should implement the `QP.gradient` and `QP.train_GD` functions. Your generated plot should be looking like the following:



Remark: Your plot is expected to demonstrate that the initialization plays an important role in the generalization performance—different initialization can lead to different global minimizers with different generalization performance. In other words, the initialization has an implicit regularization effect.

(g) **[1 point (Written)] Implicit regularization of initialization analysis**

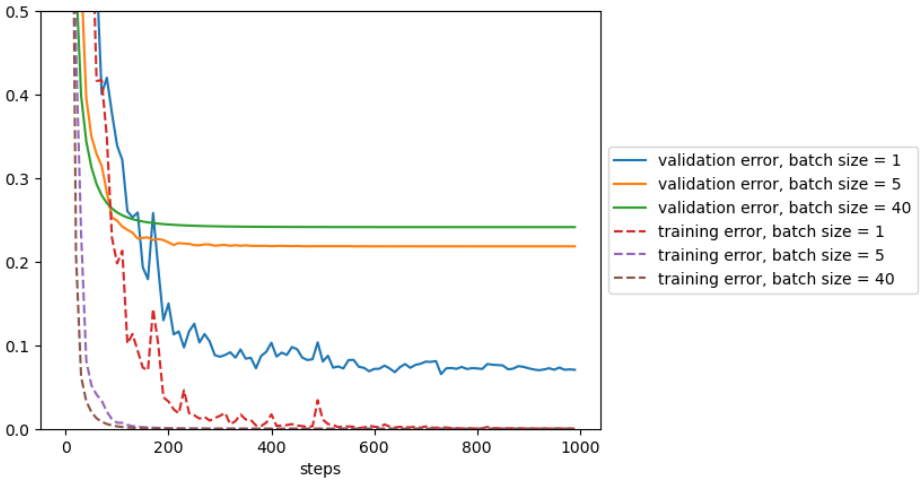
Answer the following two questions based on the plot in part (f):

- Which models can fit the training set?
- Which initialization achieves the best validation error?

(h) **[1.50 points (Coding)] Implicit regularization of batch size**

We still work with the setup in part (e). For this sub-question, we use the same dataset and starter code as in sub-question (f). We will show that the noise in the training process also induces implicit regularization. In particular, the noise introduced by *stochastic* gradient descent in this case helps generalization. **Implement** the SGD algorithm in the `QP.train_SGD` function and run it with batch size $\{1, 5, 40\}$, learning rate 0.08, and initialization $\alpha = 0.1$. For simplicity, the code for selecting a batch of examples is already provided in the starter code.

Your generated plot should like like the following:



The plot shows that the stochasticity in the training process is also an important factor in the generalization performance — in our setting, SGD finds a solution that generalizes better. In fact, a conjecture is that stochasticity in the optimization process (such as the noise introduced by a small batch size) helps the optimizer to find a solution that generalizes better. This conjecture can be proved in some simplified cases, such as the quadratically parameterized model in this sub-question (adapted from the paper HaoChen et al., 2020), and can be observed empirically in many other cases.

Files to Submit: `src-implicitreg/submission.py`

(i) **[1 point (Written)] GD vs SGD**

Compare the results with those in sub-question (f) with the same initialization. Does SGD find a better solution?

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.

2.a

$$\begin{aligned}
\ell(\theta^{(t+1)}) &= \alpha \ell_{\mathsf{sup}}(\theta^{(t+1)}) + \ell_{\mathsf{unsup}}(\theta^{(t+1)}) && \text{Definition} \\
&\geq \alpha \ell_{\mathsf{sup}}(\theta^{(t+1)}) + \sum_{i=1}^{n} \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t+1)})}{Q_i^{(t)}(z^{(i)})} && \text{Jensen's inequality} \\
&\geq
\end{aligned}$$

2.b

2.c

List the parameters which need to be re-estimated in the M-step:

In order to simplify derivation, it is useful to denote

$$w_j^{(i)} = Q_i^{(t)}(z^{(i)} = j),$$

and

$$\tilde{w}_j^{(i)} = \begin{cases} \alpha & \tilde{z}^{(i)} = j \\ 0 & \text{otherwise.} \end{cases}$$

We further denote $S = \Sigma^{-1}$, and note that because of chain rule of calculus, $\nabla_S \ell = 0 \Rightarrow \nabla_\Sigma \ell = 0$. So we choose to rewrite the M-step in terms of S and maximize it w.r.t S, and re-express the resulting solution back in terms of Σ.

Based on this, the M-step becomes:

$$\begin{aligned} \phi^{(t+1)}, \mu^{(t+1)}, S^{(t+1)} &= \arg\max_{\phi,\mu,S} \sum_{i=1}^{n} \sum_{j=1}^{k} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \phi, \mu, S)}{Q_i^{(t)}(z^{(i)})} + \alpha \sum_{i=1}^{\tilde{n}} \log p(\tilde{x^{(i)}}, \tilde{z^{(i)}}; \phi, \mu, S) \\ &= \end{aligned}$$

Now, calculate the update steps by maximizing the expression within the argmax for each parameter (We will do the first for you).

ϕ_j: We construct the Lagrangian including the constraint that $\sum_{j=1}^{k} \phi_j = 1$, and absorbing all irrelevant terms into constant C:

$$\begin{aligned} \mathcal{L}(\phi, \beta) &= C + \sum_{i=1}^{n} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j + \sum_{i=1}^{\tilde{n}} \sum_{j=1}^{k} \tilde{w}_j^{(i)} \log \phi_j + \beta \left(\sum_{j=1}^{k} \phi_j - 1 \right) \\ \nabla_{\phi_j} \mathcal{L}(\phi, \beta) &= \sum_{i=1}^{n} w_j^{(i)} \frac{1}{\phi_j} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \frac{1}{\phi_j} + \beta = 0 \\ &\Rightarrow \phi_j = \frac{\sum_{i=1}^{n} w_j^{(i)} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}}{-\beta} \\ \nabla_\beta \mathcal{L}(\phi, \beta) &= \sum_{j=1}^{k} \phi_j - 1 = 0 \\ &\Rightarrow \sum_{j=1}^{k} \frac{\sum_{i=1}^{n} w_j^{(i)} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}}{-\beta} = 1 \\ &\Rightarrow -\beta = \sum_{j=1}^{k} \left(\sum_{i=1}^{n} w_j^{(i)} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \right) \\ \Rightarrow \phi_j^{(t+1)} &= \frac{\sum_{i=1}^{n} w_j^{(i)} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}}{\sum_{j=1}^{k} \left(\sum_{i=1}^{n} w_j^{(i)} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)} \right)} \\ &= \frac{\sum_{i=1}^{n} w_j^{(i)} + \sum_{i=1}^{\tilde{n}} \tilde{w}_j^{(i)}}{n + \alpha \tilde{n}} \end{aligned}$$

μ_j: Next, derive the update for μ_j. Do this by maximizing the expression with the argmax above with respect to μ_j.

First, calculate the gradient with respect to μ_j:

$$\nabla_{\mu_j} =$$

Next, set the gradient to zero and solve for μ_j:

$$0 =$$

Σ_j: Finally, derive the update for Σ_j via S_j. Again, Do this by maximizing the expression with the argmax above with respect to S_j.

.

First, calculate the gradient with respect to S_j:

$$\nabla_{S_j} =$$

Next, set the gradient to zero and solve for S_j:

$$0 =$$

This results in the final set of update expressions:

$$\phi_j :=$$
$$\mu_j :=$$
$$\Sigma_j :=$$

2.f

3.a

3.b

4.a

4.b

4.d

4.e

4.g

4.i