EECS 189 Introduction to Machine Learning
Fall 2020

HW12

This homework is due **Tuesday, November 24 at 11:59 p.m.**

1 Getting Started

Read through this page carefully. You may typeset your homework in latex or submit neatly handwritten/scanned solutions. Please start each question on a new page. Deliverables:

1. Submit a PDF of your writeup, **with an appendix for your code**, to the appropriate assignment on Gradescope. If there are graphs, include those graphs in the correct sections. Do not simply reference your appendix.
2. If there is code, submit all code needed to reproduce your results.
3. If there is a test set, submit your test set evaluation results.

After you've submitted your homework, watch out for the self-grade form.

(a) Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions nor have I looked at any online solutions to any of these problems. I have credited all external sources in this write up.

2 Decision Trees and Random Forests

In this problem, you will implement decision trees, random forests, and boosted trees for classification on two datasets:

1. Titanic Dataset: predict Titanic survivors
2. Spam Dataset: predict if a message is spam

In lecture, you were given a basic introduction to decision trees and how such trees are learned from training data. You were also introduced to random forests. Feel free to research different decision-tree training techniques online.

Data format for Spam The preprocessed spam dataset given to you as part of the homework in `spam_data.mat` consists of 11,029 email messages, from which 32 features have been extracted as follows:

- 25 features giving the frequency (count) of words in a given message which match the following words: pain, private, bank, money, drug, spam, prescription, creative, height, featured, differ, width, other, energy, business, message, volumes, revision, path, meter, memo, planning, pleased, record, out.
- 7 features giving the frequency (count) of characters in the email that match the following characters: `;, $, #, !, (, [, &`.

The dataset consists of a training set size 5172 and a test set of size 5857.

NOTE: You should NOT use any software package for decision trees for Part b.

(a) Before applying the decision-tree learning algorithm to the Titanic dataset, we will first preprocess the dataset. In the real-world, pre-processing the data is a very important step since real-life data can be quite imperfect. However, to make this problem easier, we have provided some code to preprocess the data. **Look and the code in Jupyter Notebook Part (a) and describe how we deal with the following problems**:

- Some data points are misssing class labels;
- Some features are not numerical values;
- Some data points are missing some features.

Data Processing for Titanic Here is a brief overview of the fields in the Titanic dataset.

(a) survived - 1 is survived; 0 is not. This is the class label.
(b) pclass - Measure of socioeconomic status: 1 is upper, 2 is middle, 3 is lower.
(c) sex - Male/Female
(d) age - Fractional if less than 1.
(e) sibsp - Number of siblings/spouses aboard the Titanic
(f) parch - Number of parents/children aboard the Titanic
(g) ticket - Ticket number
(h) fare - Fare.
(i) cabin - Cabin number.
(j) embarked - Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton)

(b) **In Jupyter Notebook Part (b), implement the *information gain*, i.e., entropy of the parent node minus the weighted sum of entropy of the child nodes and *Gini purification*, i.e., Gini impurity of the parent node minus the weighted sum of Gini impurities of the child nodes splitting rules for greedy decision tree learning.**

Note: The sample implementation assumes that all features are continuous. You may convert all your features to be continuous or augment the implementation to handle discrete features.

(c) **In Jupyter Notebook Part (c), train a shallow decision tree on the Titanic and Spam dataset.** (for example, a depth 3 tree, although you may choose any depth that looks good) and **visualize your tree** (including your visualization results in your submission). Include for each non-leaf node the feature name and the split rule, and include for leaf nodes the class your decision tree would assign. You may use `sklearn` in this problem.

In Jupyter Notebook Part (c), we provide you a code snippet to draw sklearn's tree using `pydot` and `graphviz`. If it is hard for you to install these dependencies, you need to draw the diagram by hand.

(d) From this point forward, you are allowed to use `sklearn.tree.*` and the classes we have imported for you in the starter code. You are NOT allowed to use other functions from `sklearn`. **Fill in code in Jupyter Notebook Part (d) for bagged trees:** for each tree up to n, sample *with replacement* from the original training set until you have as many samples as the training set. Fit a decision tree for each sampling.

(e) **In Jupyter Notebook Part (e), apply bagged trees to the titanic and spam datasets. Find and state the most common splits made at the root node of the trees.** For example:

(a) ("thanks") < 4 (15 trees)

(b) ("nigeria") ≥ 1 (5 trees)

(f) **Read the code in Jupyter Notebook Part (f) for random forests as follows:** again, for each tree in the forest, sample *with replacement* from the original training set until you have as many samples as the training set. Learn a decision tree for each sample, this time using a randomly sampled subset of the features (instead of the full set of features) to find the best split on the data. Let `m` denote the number of features to subsample. **You do not need to answer this part, only need to go through the code written in Jupyter Notebook Part (f).**

(g) **In Jupyter Notebook Part (g), apply bagged random forests to the titanic and spam datasets. Find and state the most common splits made at the root node of the trees.**

(h) Implement the AdaBoost algorithm for a boosted random forest as follows: this time, we will build the trees sequentially. We will collect one sampling at a time and then we will change the weights on the data after each new tree is fit to generate more trees that focus their attention on tackling some of the more challenging data points in the training set. Let $w \in \mathbb{R}^N$ denote the probability vector for each datum (initially, uniform), where N denotes the number of data points. To start off, as before, sample *with replacement* from the original training set accordingly to w until you have as many samples as the training set. Fit a decision tree for this sampling, again using a randomly sampled subset of the features. Compute the weight for tree j based on its weighted accuracy:

$$a_j = \frac{1}{2} \log \frac{1 - e_j}{e_j}$$

where e_j is the weighted error:

$$e_j = \frac{\sum_{i=1}^{N} I_j(x_i) w_i}{\sum_{i=1}^{N} w_i}$$

and $I_j(x_i)$ is an indicator for data i being *incorrectly classified by this learned tree*.

Then update the weights as follows:

$$w_i^+ = \begin{cases} w_i \exp\left(a_j\right) & if I_j(x_i) = 1 \\ w_i \exp\left(-a_j\right) & \text{otherwise} \end{cases}$$

Repeat until you have M trees.

Predict by first calculating the score $z(x, c)$ for a data sample x and class label c:

$$z(x, c) = \sum_{j=1}^{M} a_j I_j(x, c).$$

where $I_j(x, c)$ is now an indicator variable for whether tree j predicts class label c for data x.

Then, the class with the highest weighted votes is the prediction (classification result):

$$\hat{y} = \arg\max_c z(x, c)$$

Fill in code in Jupyter Notebook Part (h) for boosted random forests. How are the trees being weighted? **Describe qualitatively what this algorithm is doing. What does it mean when $a_i < 0$, and how does the algorithm handle such trees? Apply boosted trees to the titanic and spam datasets.**

(i) **Run code in in Jupyter Notebook Part (i), summarize the performance evaluation of: a constant classifier, a single decision tree, a single decision tree (sklearn-implementation), bagged trees, random forests, and boosted trees**. For each of the 2 datasets, report your training and validation accuracies. You should use a 3-fold cross validation, i.e., splitting the dataset into 3 parts. You should be reporting 32 numbers (2 datasets $\times$ 4 classifiers $\times(3 + 1)$ for 3 cross validation accuracies and 1 training accuracy). Describe qualitatively which types of trees and forests performed best. Detail any parameters that worked well for you. **In addition, for each of the** 2 **datasets, train your best model and submit your predictions on the test data to Gradescope**. Your best Titanic classifier should exceed 73% accuracy and your best Spam classifier should exceed 76% accuracy for full points.

(j) **For the spam dataset only: Go through the Jupyter Notebook Part (j), given examples shown in Jupyter Notebook Part (j), describe what kind of data are the most challenging to classify and which are the easiest. Also, you could come up with your own procedure for determining which data are easy or hard to classify.**

3 NLP and embeddings

Common Natural Language Processing (NLP) tasks involve spell-checking (easy), information extraction (medium), and machine translation (hard). A common first step in any NLP task is the representation of words. A vector representation of a word is called an *embedding*. Initially, words were represented as one-hot vectors in same way that we described representing other categorical data. If the size of the dictionary is $|V|$ (For reference, there are about 30K distinct words in the complete works of Shakespeare), the word `school` would be represented as

$$0, 0, \ldots, 0, 1, 0, \ldots, 0,$$

where the 1 happens at the index corresponding to the word `school`. A document d can be summarized as the mean of the vectors corresponding to all its words. This is called the *bag-of-words* model because it throws away the information of the relationship between words (similar to the relation between an image and its histogram). When we have a vector representation for a document, we can check how similar two documents d, d' are by computing the cosine of the angle between them.

We know words like `energy` and `energize` are related, but one-hot encoding hides all similarity between words (the inner product between any two different one-hot representations is zero). A different way to create embeddings was proposed in the 1950s: words with similar meanings appear in similar contexts. This connection between meaning and distribution is called the *distributional hypothesis*. One of the early workers in the field, J. R. Firth, stated this idea sharply: "you shall know a word by the company it keeps." In this problem, we will study a method for learning *vector semantics* (i.e., embeddings learned under the distributional hypothesis) called word2vec.

word2vec learns dense encodings (short vectors) for words in a self-supervised fashion: all the algorithm needs is running text. We assume that every word w in the vocabulary has a pair of embeddings $\phi(w), \psi(w) \in \mathbb{R}^d$, where d is the fixed length of the learned vectors (a hyperparameter usually ranging from 50 to 200). Suppose we have another word w' with embeddings $\phi(w')$ and $\psi(w')$. Then word2vec operates under the claim that the probability that w' is a *context word* for w is

$$P(w' \texttt{ is a context word for } w) = \frac{1}{1 + e^{-\psi(w')^\top \phi(w)}}.$$

To be a context word means that w' tends to appear close to w in written text. This will be clearer soon.

(a) Suppose we have a large dataset $\{(w_i, w_i', t_i')\}_{i=1}^N$ of pairs of words w_i, w_i' together with a label t_i such that $t_i = 1$ when w_i' is a context word for w_i and $t_i = 0$ when it is not. **Assuming all pairs of words are independently drawn (completely unfounded!), give an expression for the log likelihood $\ell(t_1, \ldots, t_N | w_1, w_1', \cdots, w_N, w_N'; \phi(w_1), \psi(w'), \ldots, \phi(w_N), \psi(w_N))$.**

(b) Observe that the log likelihood uses two embeddings $\phi(w_i), \psi(w_i)$ per word w_i: $\phi(w_i)$ is used when w_i is the *center word*, and $\psi(w_i)$ is used when w_i is a context word for another center

word. We can try to learn the embeddings by maximizing the likelihood, and we can do that by running SGD on the negative log likelihood. **Provide an expression for the SGD updates to the weights $\phi(w_i), \psi(w_i)$ based on the log likelihood.**

(c) We know how to update the vectors $\phi(w), \psi(w)$ for a word w based on the logistic loss as we did in the previous part. But where do we get the training set? We don't need labeled text, just written text. Consider the sentence

"Concerned with unprecedented volatility, investors shun contracts and favor stocks."

Choose `investors` as the center word. We create a window of length L (hyperparameter) around the center word. This gives us $2L$ pairs (`investors`, w', 1). For instance, for $L = 2$, $w' \in$ {`unprecedented, volatility, shun, contracts`}. By moving the center word in a corpus of text, we generate a large dataset of labeled pairs. Note, however, that this procedure would only yield positive examples. To learn effectively, we also need some negative examples for self-supervision. Negative examples are usually generated by choosing at random words from the dictionary according to their probability of occurrence in the string. (Why? Because that is what their frequency would be in positive examples so we want to match that in negative examples for balance.) **In the jupyter notebook, complete the code for the updates to the word embeddings.** Training the embeddings may take up to one hour. Note the strings we are using come from transcripts of the European parliament. Source: http://www.statmt.org/europarl/.

(d) The code of this section generates the list of the five words which are closest (in angle) to an input word. **Browse the values of the vocabulary and find two words whose closest words sound mostly reasonable to you. Then find another two words whose closest words you deem semantically objectionable. In both cases, state the words you used, their closest words, and your thoughts about their relatedness.** *Hint: The input we are using to learn the embeddings comes from transcripts from the European parliament. It's probably wiser to choose words common in diplomatic settings.*

(e) Look at the way in which we trained our vectors in part C of the jupyter notebook. There is no clearly distinguishable training set and validation set. **Propose a way in which we can convert our data into a train+validation set and suggest a metric for checking the quality of our embeddings.**

This is an open-ended question, but it is important for you to be able to think in this way in practice. Try.

(f) Suppose we have learned d-dimensional embeddings for a vocabulary V. Then we have $|V| \times d$-matrices Φ and Ψ containing the center and context embeddings, respectively, for our words. Suppose we add a new word w' to our vocabulary. We want to learn the embeddings of w' while keeping the existing embeddings unmodified. In the jupyter notebook, you will see that we extracted all positive examples for a word w in the vocabulary V and we claimed that a new word w' has exactly the same positive examples (this is like doing search replace in a text). **Complete the code of this section to train the new center embedding and report the 5 words with embeddings closest to the embedding for w'. Do the results surprise you?**

(g) Realistic semantic embeddings take over 10 hours to train. We will keep exploring embeddings by loading previously-trained embeddings. We downloaded the 50-dimensional GloVe (another algorithm) vectors from https://nlp.stanford.edu/projects/glove/ and loaded them in the jupyter notebook. The code for this part plots in 2D the PCA projections of the vectors for several words. **Choose 10-20 words and plot the 2D projections of the embeddings. What do you notice? Which words are close to each other? Which words are farther away?**

(h) It has been found that GloVe vectors can be used to solve some analogies. Given three words, w, w', w'', we compute $\phi(w) - \phi(w') + \phi(w'')$, and we observe that the resulting vector can sometimes give the answer to an analogy of the form *w' is to w what w'' is to ???*. **Using the example code, find three analogies for which the code gives a sensical answer, and three analogies for which it does not. In the nonsensical case, explain what you expected.**

What's next? When trying to understand a sentence in English, it is important to figure out how words relate to each other grammatically (what is the object, subject...). This goes by the name dependency parsing, and it's a necessary task for translation and question answering. If you are Amazon's Alexa, and you are told to do something, you must understand how the words spoken to you relate to entities in the *real world* and to actions you can take (e.g., play a song or turn lights on). This and more you will study in an NLP class. For now, if you want to get more of a taste for this area, go and read something about GPT-3; this will acquaint you with cutting-edge results in many NLP tasks.

4 Adversarial Examples (Theory)

Many machine learning models are known to be sensitive to small perturbations on the input. This phenomenon, dubbed *adversarial examples*, is an on-going area of research in machine learning, and is viewed as a particularly challenging problem for many deep learning contexts. In the previous homework, we have shown that many classifiers are vulnerable to adversarial examples and that there is a natural way to try to "robustify" these classifiers. This problem is a sequel and focuses on theoretical analysis in a simple setting. This example illustrates why the adversarial example phenomenon should perhaps be expected and also has a kind of inevitability to it.

For this problem, we will consider data with input $\mathbf{x}$ and binary label $y \in \{+1, -1\}$. We assume that the class is balanced, i.e. $P(y = +1) = P(y = -1) = 1/2$. We are only concerned with linear classifiers of the form: $f(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x})$.

(a) For now, let's assume that the features $\mathbf{x}$ have the following distribution.

$$x_i \sim \mathcal{N}(\eta y, \sigma^2) \quad \forall i \in \{1, \ldots, d\}$$

With different features being independent conditioned on the label y. In other words, all features of $\mathbf{x}$ are all weakly correlated with the label y. A natural choice for a linear classifier in this case is to choose a uniform weight over all features, i.e. $w_{unif} = [\frac{1}{d}, \frac{1}{d}, \ldots, \frac{1}{d}]$. For this

choice of the classifier $f(\mathbf{x}) = \text{sign}(\mathbf{w}_{unif}^\top \mathbf{x})$ to achieve 99% accuracy on the test distribution i.i.d. drawn from the same distribution above, **what is approximately the smallest value of η in terms of σ and d?**

(Hint: Look at the distribution for the random variable representing $\mathbf{w}_{unif}^\top \mathbf{x}$. What is it? What has to be true to have accuracy 99%?)

(b) The result from part (a) should suggest that η can be fairly small especially when d is large, and so it is easy to classify such data correctly with high accuracy.

Now consider a slightly different setting where we will test the same classifier against adversarial examples. **For a given sample $(\mathbf{x}, y)$, derive adversarial examples $\mathbf{x}_{adv}$ for the classifier from part (a) where the ℓ_∞-norm of the perturbation is contrained to ϵ, i.e. $\|\delta\|_\infty \leq \epsilon$. Then, argue why this classifier will have very low accuracy on the adversarial examples if the adversary can use $\epsilon \geq 2\eta$.**

(Hint: To derive the adversarial examples, you can use the result from the previous homework.)

(c) Use the same setting and the adversarial examples from part (b), and the assumption that the classes are balanced in this problem. **What is the highest adversarial accuracy we can hope to achieve for any classifier with an adversary that powerful?** Briefly justify your answer. No proof is needed for this part.

(d) For this part only, let's consider a slightly different distribution for $\mathbf{x} \in \mathbb{R}^{d+1}$.

$$x_1 = y, \quad x_i \sim \mathcal{N}(\eta y, \sigma^2) \quad \forall i \in \{2, \ldots, d+1\}$$

Assume that $\eta = \frac{4\sigma}{\sqrt{d}} < 1$.

We want to understand how robust can a classifier be to adversarial perturbations given this kind of input data.

Suppose that we insist that we do better than guessing for any $\epsilon = 2\eta < \bar{\epsilon}$. **How big can $\bar{\epsilon}$ be in this case?**

Then think about what the most robust classifier is (i.e. what the weights of the classifier should look like if you want to achieve the highest accuracy on the adversarial examples?). You should realize that you want to prioritize the first feature, and the robust weights should have the following form:

$$\mathbf{w}_{rob} = [\alpha, \beta, \beta, \ldots, \beta] \in \mathbb{R}^{d+1} \qquad \text{for } \alpha > 0$$

What is the range of values β such that this robust classifier has perfect accuracy on the adversarial examples? Answer in terms of α, ϵ, d. **Which value of β gives you the largest margin?**

(e) *(Trade-off between normal and adversarial accuracy.)* Now consider the following data distribution which is slightly modified from the previous part and is a bit more realistic. For $p > 0.5$,

$$x_1 = \begin{cases} +y & \text{w.p.} \quad p \\ -y & \text{w.p.} \quad 1-p \end{cases}, \quad x_i \sim \mathcal{N}(\eta y, \sigma^2) \quad \forall i \in \{2, \ldots, d+1\}$$

Your classifier from the previous part should have a perfect accuracy on the normal test set from the previous part, but that is because the features of $\mathbf{x}$ in the previous part are too easy. In practice, we are more likely to encounter a set of features where most of them are noisy and only weakly correlated with the true label, and only a few features are strongly correlated but still with some error. In this case, we choose the first feature to be the only "strong" feature and the rest d features to be the "weak" ones.

Once again suppose $\eta = \frac{4\sigma}{\sqrt{d}} < 1$ and $\epsilon = 2\eta < \bar{\epsilon}$ from the previous part. **Argue that we may not be able to have a single linear classifier that is both robust and has high accuracy on un-perturbed data at the same time.**

Hint: What are the normal and the adversarial accuracies of $\mathbf{w}_{unif}$ *and* $\mathbf{w}_{rob}$*?*

5 Meta-learning for Learning 1D functions

We will return to the by-now familiar setup of learning a 1D function from Fourier features in an overparameterized setting, introduced in Problem 4 of Homework 5. In the past we worked with a version in which the first s features were known to be useful and thus assigned larger weights before performing regression. Suppose now that our task is not to learn not just one 1D function, but any of a class of 1D functions drawn from a *task distribution* $\mathcal{D}_T$.

You will notice a spiritual connection between this problem and the earlier homework problem on learning dimensionality reduction using auxiliary labels. This is not a coincidence and we hope this problem will help you better appreciate the interconnected nature of machine learning concepts and the power of actually understanding them from the foundations.

In this problem we consider all signals of the form

$$y = \sum_{s \in \mathcal{S}} \alpha_s \phi_s^u(x)$$

The task distribution produces individual tasks which have true features with random coefficients in some *a priori* unknown set of indices $\mathcal{S}$. Although we do not yet know the contents of $\mathcal{S}$, we can sample tasks from $\mathcal{D}_T$. The important question is thus, how do we use sampled tasks in training to improve our performance on an unseen task drawn from $\mathcal{D}_T$ at test time? One solution is to use our training tasks to learn a set of weights to apply to the features before performing regression through meta-learning.

In the original problem formulation, we chose feature weights c_k to apply to the features $\phi_k^u(x)$ before learning coefficients $\hat{\beta}_k$ such that

$$\hat{y} = \sum_{k=0}^{d-1} \hat{\beta}_k c_k \phi_k^u(x).$$

We then performed the min-norm least-squares optimization

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \|\boldsymbol{\beta}\|_2^2$$

$$\text{s.t. } \mathbf{y} = \sum_{k=0}^{d-1} \beta_k c_k \mathbf{\Phi}_k^u$$

where $\mathbf{\Phi}^u$ is the column vector of features $[\phi_0^u(x), \phi_1^u(x), \ldots, \phi_{d-1}^u(x)]'$ which are orthonormal with respect to the test distribution. In our new formulation we want to learn $\mathbf{c}$ which minimizes the expectation for $\hat{\boldsymbol{\beta}}$ over all tasks,

$$\arg\min_{\mathbf{c}} \mathbb{E}_{\mathcal{D}_T}\left[\mathcal{L}_T\left(\hat{\boldsymbol{\beta}}_T, \mathbf{c}\right)\right]$$

where $\mathcal{L}_T\left(\hat{\boldsymbol{\beta}}_T, \mathbf{c}\right)$ is the loss from learning *bb*$\hat{e}$*ta* for a specific task with the original formulation and a given $\mathbf{c}$ vector. $\mathbf{c}$ is shared across all tasks and is what we will optimize with meta-learning.

There are many machine learning techniques which can fall under the nebulous heading of meta-learning, but we will focus on one with Berkeley roots called Model Agnostic Meta-Learning (MAML)[1] which optimizes the initial weights of a network to rapidly converge to low loss within the task distribution. The MAML algorithm as described by the original paper is shown in Fig. 1. This problem should help you understand how and why MAML works, without any mystification.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Figure 1: MAML algorithm. We will refer to the training steps on line 6 as the *inner update*, and the training step on line 8 as the *meta update*.

At a high level, MAML works by sampling a "mini-batch" of tasks $\{T_i\}$ and using regular gradient descent updates to find a new set of parameters θ_i for each task starting from the same initialization θ. Then the gradient w.r.t. the original θ each calculated for each task using the task-specific updated weights θ_i, and θ is updated with these 'meta' gradients. Fig. 2 illustrates the path the weights take with these updates.

[1] C. Finn, P. Abbeel, S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," in *Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017*

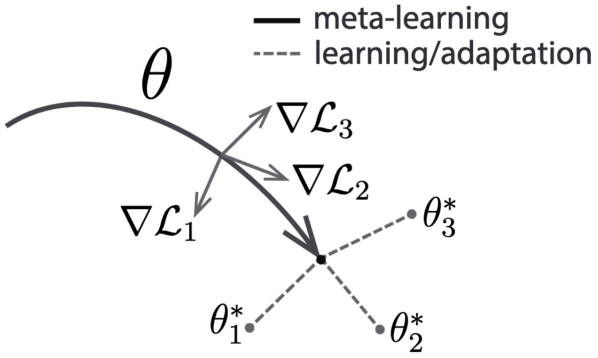

Figure 2: MAML gradient trajectory illustration

The end goal is to produce weights θ^* which can reach a state useful for a particular task from $\mathcal{D}_T$ after a few steps — needing to use less data to learn. If you want to understand the fine details of the algorithm and implementation, we recommend reading the original paper and diving into the code provided with this problem.

(a) For now, assume that we have access to grid-spaced training data *and* we have M alias groups and $|\mathcal{S}| = s < M$ indices (max one feature from each alias group) that are actually present in any of the true signals that we are going to encounter. **Argue that if the feature weights on the $s \in \mathcal{S}$ indices are large compared to the rest then regression tasks will work.**

(Hint: Can you relate this to the familiar setting from the previous problem? Think about the differences in this setup and argue why they shouldn't matter for the success of regression given the necessary assumptions in the previous problem.)

(b) For utter simplicity, suppose now that we have exactly one training point (x, y), one true feature $\phi_t^u(x)$, and one alias of the true feature $\phi_a^u(x)$ that agrees with the true feature on this one point, even though both features are orthonormal relative to the test distribution. The function we wish to learn is $y = \phi_t^u(x)$. We learn coefficients $\hat{\boldsymbol{\beta}}$ using the training data. Note, both the coefficients and the feature weights are 2-d vectors. The first component corresponds to the true feature and the second one to the alias.

Derive the closed-form solution for $\hat{\boldsymbol{\beta}}$ with grid-spaced training data and feature weights c_0, c_1.

(c) Assume for simplicity that we have access to infinite data from the test distribution for the purpose of updating the feature weights $\mathbf{c}$. **Calculate the gradient of the expected test error with respect to the true and alias feature weights c_0 and c_1, respectively:**

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{c}}\left(\mathbb{E}_{x_{test},y_{test}}\left[\frac{1}{2}\left\|y-\hat{\beta}_0 c_0\phi_t^u(x)-\hat{\beta}_1 c_1\phi_a^u(x)\right\|_2^2\right]\right).$$

(Hint: the features $\phi_i^u(x)$ are orthonormal under the test distribution.)

(d) **Generate a plot showing that, for some initialization $\mathbf{c}^{(0)}$, as the number of iterations $i \to \infty$ the weights empirically converge to $c_0 = \|\mathbf{c}^{(0)}\|$, $c_1 = 0$ using gradient descent with**

a sufficiently small step size. Include the initialization and its norm and the final weights. What will β go to?

The rest of the parts (e)-(k) are in the accompanying jupyter notebooks 'prob2-part1.ipynb' and 'prob2-part2.ipynb'. Follow the instructions in the notebook and write your answers in the cells provided.

(e) Meta-learning with closed-from min-norm least squares regression, all uniform random data.

(f) Impact of incorrect feature weightings.

(g) Meta-learning with closed-from min-norm least squares regression, grid-spaced training data and uniform random meta update and test data.

(h) Replacing the closed-form solution with gradient descent.

(i) Impact of the number of gradient descent steps on meta-learning success.

(j) Meta-learning for classification with copy-pasted regression code.

(k) Meta-learning for classification with logistic regression.

6 Your Own Question

Write your own question, and provide a thorough solution.

Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

Contributors:

- Ana Tudor
- Anant Sahai
- Ashwin Pananjady
- Cathy Wu
- Chawin Sitawarin
- Inigo Incer
- Josh Sanz
- Mandi Zhao
- Mark Velednitsky
- Vignesh Subramanian
- Yang Gao
- Yaodong Yu
- Yichao Zhou