MATH 342W / 650.4 Spring 2022 Homework #4

Peter Antonaros

Due 11:59PM April 14 by email

(this document last updated 12:21am on Friday 15th April, 2022)

Instructions and Philosophy

The path to success in this class is to do many problems. Unlike other courses, exclusively doing reading(s) will not help. Coming to lecture is akin to watching workout videos; thinking about and solving problems on your own is the actual "working out." Feel free to "work out" with others; **I want you to work on this in groups.**

Reading is still *required.* You should be googling and reading about all the concepts introduced in class online. This is your responsibility to supplement in-class with your own readings.

The problems below are color coded: green problems are considered *easy* and marked "[easy]"; yellow problems are considered *intermediate* and marked "[harder]", red problems are considered *difficult* and marked "[difficult]" and purple problems are extra credit. The *easy* problems are intended to be "giveaways" if you went to class. Do as much as you can of the others; I expect you to at least attempt the *difficult* problems.

This homework is worth 100 points but the point distribution will not be determined until after the due date. See syllabus for the policy on late homework.

Up to 7 points are given as a bonus if the homework is typed using LaTeX. Links to instaling LaTeX and program for compiling LaTeX is found on the syllabus. You are encouraged to use `overleaf.com`. If you are handing in homework this way, read the comments in the code; there are two lines to comment out and you should replace my name with yours and write your section. The easiest way to use overleaf is to copy the raw text from hwxx.tex and preamble.tex into two new overleaf tex files with the same name. If you are asked to make drawings, you can take a picture of your handwritten drawing and insert them as figures or leave space using the "\vspace" command and draw them in after printing or attach them stapled.

The document is available with spaces for you to write your answers. If not using LaTeX, print this document and write in your answers. I do not accept homeworks which are *not* on this printout. Keep this first page printed for your records.

NAME: ______________Peter Antonaros______________

Problem 1

These are some questions related to polynomial-derived features and logarithm-derived features in use in OLS regression.

(a) [harder] What was the overarching problem we were trying to solve when we started to introduce polynomial terms into $\mathcal{H}$? What was the mathematical theory that justified this solution? Did this turn out to be a good solution? Why / why not?

The overarching problem we were trying to solve was the reduction of misspecification error in our model. Since we cannot combat error due to ignorance, and our algorithms ensured very low estimation error, the only way to make our model better was buy emplyoing a richer H space. This idea is based in the Weierstrass Approximation Theorem which proves that any continuous function f can be approximated for $x \in [a, b]$ with arbitrary precision by using some nth degree polynomial. Of course, under our conditions, we do not want n to be "too large" as to reduce to the chance of over fitting, but nevertheless this theorem is what allows us to use this idea in our modeling tasks.

(b) [harder] We fit the following model: $\hat{\boldsymbol{y}} = b_0 + b_1 x + b_2 x^2$. What is the interpretation of b_1? What is the interpretation of b_2? Although we didn't yet discuss the "true" interpretation of OLS coefficients, do your best with this.

We can interpret the two coefficients jointly by saying that $(b_1 + b_2 x)$ is equal to the slope of our yHat at a given value of x

(c) [difficult] Assuming the model from the previous question, if $x \in \mathcal{X} = [10.0, 10.1]$, do you expect to "trust" the estimates b_1 and b_2? Why or why not?

This is dependent on the entire interval in which our function is defined on. Assuming the entirety of our function could fit within the bounds of $[10.0, 10.1]$, then I would trust the estimated for b_1 and b_2. Otherwise, if our function is defined on an interval outside of the one stated in the question then we are essentially approximating a line on the interval. Since we have a polynomial term in our function, then this would result in a higher **order** of error.

(d) [difficult] We fit the following model: $\hat{\boldsymbol{y}} = b_0 + b_1 x_1 + b_2 \ln(x_2)$. We spoke about in class that b_1 represents loosely the predicted change in response for a proportional movement in x_2. So e.g. if x_2 increases by 10%, the response is predicted to increase by $0.1 b_2$. Prove this approximation from first principles.

When we take a neighborhood of $\Delta 1$ in our ln function, then what we are essentially doing is mimicking percent increase. This can be show by...

ln(x+1)≈ $\sum_{i=1}^{\infty} \frac{x^i}{i}(-1)^{i+1} \rightarrow$ ln(x) = ln((x+1)-1)≈x-1

With this we can say that holding x_1 constant and taking the change in x_i to be $x_{i\alpha} - x_{i\beta}$

$\Delta\hat{y} = b_2(ln(x_{2\alpha}) - ln(x_{2\beta})) = b_2(ln(\frac{x_{2\alpha}}{x_{2\beta}})) \approx b_2(\frac{x_{2\alpha}}{x_{2\beta}} - 1)$, which we can interpret as the percent increase.

(e) [easy] When does the approximation from the previous question work? When do you expect the approximation from the previous question not to work?

As mentioned previously this works when the change in x is approximately 1, in either the positive or negative direction. I would not expect this to work in the function from the previous question if we are comparing x_i's whose Δ is not approximately equal to one.

(f) [harder] We fit the following model: $\ln(\hat{\boldsymbol{y}}) = b_0 + b_1 x_1 + b_2 \ln(x_2)$. What is the interpretation of b_1? What is the *approximate* interpretation of b_2? Although we didn't yet discuss the "true" interpretation of OLS coefficients, do your best with this.

(g) [easy] Show that the model from the previous question is equal to $\hat{\boldsymbol{y}} = m_0 m_1^{x_1} x_2^{b_2}$ and interpret m_1.

$\hat{y} = e^{ln\hat{y}} = e^{b_0 + b_1 x_1 + lnb_2 x_2} = e^{b_0}(e^{b1})^{x_1}(e^{lnx_2})^{b_2}$

Then we can say with $m_0 = e^{b_0} and m_1 = e^{b_1}]$

$\hat{y} = m_0 m_1^{x_1} x_2^{b_2}$

Problem 2

These are some questions related to extrapolation.

(a) [easy] Define extrapolation and describe why it is a net-negative during prediction.

Extrapolation as it is defined in machine learning is the act of using new data which is outside the range of our training data. We are no longer predicting by use of inputs within a range (interpolation), but attempting to predict outside this same range (extrapolation). It is net negative simply because our models can cease to make sense outside this range. Taking the most basic example, a linear model to predict housing cost based on home square footage. Our data ranges from 1000 square feet to 10000 square feet, and so if we were to extrapolate, what price would be expect for a house with negative square feet, perhaps a negative price? The model ceases to make sense as it really only exists within this known range (1000-10000 square feet). This would be the empirical reasoning for why extrapolation is net negative. Analytically, to compute error we do so against known observation points. If we are attempting to predict with values for our features outside this range, we have little to nothing to compute error with. The model can perform well, or more likely it will perform wildly, and we have no way of gauging this.

(b) [easy] Do models extrapolate differently? Explain.

Different models will extrapolate differently. This can be explained by referring to Runge's Polynomial Phenomena, where high degree polynomial approximations will have a higher degree of oscillation at the endpoints. This occurs because we essentially run out of data to interpolate the polynomial and it "behaves" as it wishes outside the known input range. To mitigate this we can use lower degree polynomial models, thereby changing how the models extrapolate. Of course this is not limited to Polynomials, but it is easiest to demonstrate the idea with them.

(c) [easy] Why do polynomial regression models suffer terribly from extrapolation?

As mentioned in the previous answer, polynomial regression models suffer terribly from extrapolation due to their high rate of change, which is often times amplified at the endpoints. This amplification of change at the endpoints is worse than we would expect since this is the area where extrapolation begins. High order polynomials perform even worse, since they give us less "room" outside these endpoints before our function goes off to its limits. Lower order O-Notation functions will also suffer from extrapolation, but because they change "slower" we must move farther away from the endpoints to reach equivalent error of high degree polynomials. In general, we should aim to stay away from extrapolation regardless of the foundational function of our model.

Problem 3

These are some questions related to validation.

(a) [easy] Assume you are doing one train-test split where you build the model on the training set and validate on the test set. What does the constant K control? And what is its tradeoff?

The constant K is useful when we think of it in terms of $\frac{1}{K}$ represents the proportion of test to train data. $K = 5$, implies 20% of our data is reserved for testing, $K = 10$ implies 10% and so on. The trade off of altering values of K is, do we want a better trained model with less of an idea of how it will perform in the future, or a worse trained model with more of an idea of how it will perform in the future. We can say that altering the K value changes our certainty bound of prediction. Lower values of K will result in a better model prediction in sample, but with less confidence that our real response out of sample will be below this. Lower values of K will result in worse model prediction in sample, but with more confidence that our real response out of sample will be below this. To restate this in another way, changing K changes the tightness of our bound. Would you rather expect to pay 50 dollars and then suddenly have to pay 75, or expect to pay 100 and really only pay 75. The choice is yours.

Generally, we choose $K = 3$, $K = 5$, $K = 10$, but since K is a hyper parameter it can be tuned, and is also specific to the application to the model.

(b) [harder] Assume you are doing one train-test split where you build the model on the training set and validate on the test set. If n was very large so that there would be trivial misspecification error even when using $K = 2$, would there be any benefit at all to increasing K if your objective was to estimate generalization error? Explain.

Given that misspecification error is low, knowing error due to ignorance is hard to defeat, and assuming we are using a good algorithm that ensures estimation error is low, then there would not be much of a point to increase K. We are doing one train-test split, and so because our misspecification error is already low, that means the subset of n reserved for training was large enough for our model to be accurate. Generalization error will follow suit as our 3 sources of error are minimal.

(c) [easy] What problem does K-fold CV try to solve?

K-Fold Cross Validation tries to solve the problem of variation between model performance when choosing a random sample of test-train split from our $\mathbb{D}$. The issue is the random sampling itself, which ensures we are not cheating by tuning the model to the data, but also results in different error metrics each time we run the model. To speak in a more statistically concrete way, K-Fold CV tries to concentrate the distribution of error in our model, so that we are more confident in its performance.

(d) [E.C.] Theoretically, how does K-fold CV solve it?

Briefly touched upon on in the previous answer, theoretically K-Fold CV tightens the probability distribution of our error around our mean. Rather than saying it improves performance, a more accurate description would be in increases our confidence in the model performance. By taking many folds of our data we are lowering the deviation of error observations, and so we have a tighter confidence interval to describe how our model will perform within. If we were to compute the area under our distribution, we could say something to the effect of "90% of the time our model performs within [x1,x2] error". Without CV this statement would look more like "This time we ran the model it performed with error metric x1".

Problem 4

These are some questions related to the model selection procedure discussed in lecture.

(a) [easy] Define the fundamental problem of “model selection”.

The fundamental problem of model selection is that we are trying to compare different models, relative to each other while at the same time ensuring that the metrics we judge them by are not down to "dumb luck". *This is why we do nested validation*

(b) [easy] Using two splits of the data, how would you select a model?

Declare one split the training set and the other split the validation set. Train the models on the training set, test them on the test sets and compare them with some error metric(s). The model with the best performance relative to the test sets would then be chosen.

(c) [easy] Discuss the main limitation with using two splits to select a model.

The main limitation of using two splits to select a model is that iif we choose to go back and tweak any of of the models, then we are essentially fitting to the data because we have now seen it all. This is not necessarily a bad thing, but when we become overly reliant on this to arbitrarily improve our model, we will suffer in generalization.

(d) [easy] Using three splits of the data, how would you perform model selection?

We split the data into train and select, and test block. We train the models on the train set, test them individually on the select blocks. Here we can do our tweaking and whatever we prefer. Finally once we are completely done with this process, we test our models on the never before seen or used *real* test block and compare error metrics.

(e) [easy] How does using both inner and outer folds in a double cross-validation nested resampling procedure improve the model selection procedure?

It improves the model selection procedure because it reduces the variance of error of each individual model, makes it much more difficult to fit the models to the data, and ensures that each model gets a "fair" chance at proving itself out of sample.

(f) [easy] Describe how g_{final} is constructed when using nested resampling on three splits of the data.

g is trained on the train block, it is tested on the *select* block and tweaked. Finally we used the actual test block to evaluate the final out of sample performance.

Once we have these out of sample metrics, we train g on the entire dataset and consider it to be g_{final}. This final model is guaranteed to perform either equally or better, indicating that our previously computed metrics should be upper error bounds.

(g) [easy] Describe how you would use this model selection procedure to find hyperparameter values in algorithms that require hyperparameters.

You would create a candidate set of hyper-parameters, h, and for each repeat the process above. Eventually this will lead you to the most optimal model with the most optimal hyper parameter tuned to the respective model.

(h) [difficult] Given raw features $x_1, \ldots, x_{p_{raw}}$, produce the most expansive set of transformed p features you can think of so that $p \gg n$.

By expansive I take this to mean the overall diversity of the functions used. In this way they are the most expansive since they cover the largest amount of possible transformations.

The most expansive set would be the set of all non linear transforming functions.

(i) [easy] Describe a methodology that can create a linear model on a subset of the transformed features (from the previous problem) that will not overfit.

Problem 5

These are some questions related to the CART algorithms.

(a) [easy] Write down the step-by-step $\mathcal{A}$ for regression trees.

Regression Trees Algorithm

1) All features and observations $< \mathcal{X}, \vec{y} >$

2) Create bins for every split $< x_l, \bar{y}_l >$, $< x_r, \bar{y}_r >$

3) Locate the split with the minimal weighted average SSE

4) Assign the respective $\bar{y}$ as our $\hat{y}$ for each corresponding bin.

5) Repeat steps 1-4 **recursively** for both sides of the split, with the stopping conditions N_0. This is a hyper parameter representing the number of observations in the leaf nodes.

(b) [difficult] Describe $\mathcal{H}$ for regression trees. This is very difficult but doable. If you can't get it in mathematical form, describe it as best as you can in English.

Describing $\mathcal{H}$ for regression trees is best explained as a tree diagram. Given some root node, with children to the left whose data is less than their parent and children to the right whose data is greater than their parent. We buildup our $\mathcal{H}$, by linking an indicator function at each node's data in the order of our nodes, iteratively obtaining our $\mathcal{H}$.

Mathematically we would define this as where $\vec{w_i}$ represents the nodes corresponding weight value, and our α_i's represent the nodes entities.:

$\mathcal{H} = \vec{w_1}\mathbb{1}_{x \leq \alpha_1} + \vec{w_2}\mathbb{1}_{x \in (\alpha_1, \alpha_2)} + \ldots + \vec{w_p}\mathbb{1}_{x > \alpha_p}$

Unfortunately this does not capture every permutation of our tree. Since we can swap nodes around, yielding a different tree who stills obeys the rules described above. For this what we can do is first establish some base tree full of all required nodes.

We can flatten the tree to an array and consider this to be a set. Since sets technically handle duplicate elements, we can subscript each flattened node to indicate the although it may have equal value to another node, fundamentally they represent values for unique features. We now have an idea of how to do this, with the main issue being how to express this mathematically in conjunction with our indicator function for each node and our weight vector $\vec{w}$.

My best attempt at actually expressing this is...

$\vec{W}$ represents the collective weights for respective nodes

$\vec{\mathbb{1}}$ represents the collective indicators for respective nodes

$S(T)$ represents the symmetrical groups of our tree node set T

$\mathcal{H} = \vec{W}^T(\vec{\mathbb{1}}^T_{x_i \leq \alpha_i : i=1, x_i \in (\alpha_i \ldots \alpha_1) : i < length(T), x_i >= \alpha_i : i = length(T)} S(T)) : T = (x_{i1}, x_{i2}, \ldots x_{ip})$

(c) [harder] Think of another "leaf assignment" rule besides the average of the responses in the node that makes sense.

Other leaf assignment rule besides the average of y in the node that makes sense would be...

Using the simple average of SSE for the right and left side of the split rather than the weighted average. I see why this isn't used as weighted average is simply superior *ba dum tss*

(d) [harder] Assume the y values are unique in $\mathbb{D}$. Imagine if $N_0 = 1$ so that each leaf gets one observation and its $\hat{\boldsymbol{y}} = y_i$ (where i denotes the number of the observation that lands in the leaf) and thus it's very overfit and needs to be "regularized". Write up an algorithm that finds the optimal tree by pruning one node at a time iteratively. "Prune" means to identify an inner node whose daughter nodes are both leaves and deleting both daughter nodes and converting the inner node into a leaf whose $\hat{\boldsymbol{y}}$ becomes the average of the responses in the observations that were in the deleted daughter nodes.

This is an example of a "backwards stepwise procedure" i.e. the iterations transition from more complex to less complex models.

We can take a top down approach to pruning the tree... Let us have a function definition of prune which,

```
prune(Node, level){
    //True if left child is leaf, false otherwise
    leftLeaf = checkChild(Node.left);
    //True if right child is leaf, false otherwise
    rightLeaf = checkChild(Node.right);

    if(leftLeaf AND right left){
        return avg(Node.left.data, Node.right.data);
    }
}
```

takes in a node and ensures that the child nodes are both leaves, returns the the average of responses in the child nodes, and finally deletes them.

We can do a level order traversal

```
if tree is NULL then return;
if level is 1, then
    prune(Node);
else if level greater than 1, then
    prune(Node.left, level-1);
    prune(Node.right, level-1);
```

(e) [difficult] Provide an example of an $f(\boldsymbol{x})$ relationship with medium noise δ where vanilla OLS would beat regression trees in oos predictive accuracy. Hint: this is a trick question.

There is no such example, because at worst Regression Trees perform equivalently to vanilla OLS. These would be in the strictly linear problems where this occurs. Here we tend to stick with vanilla OLS because it is simply easier to implement and easier computationally. For non linear problems or problems with heavy co-linearity, Regression trees will perform better.

(f) [easy] Write down the step-by-step $\mathcal{A}$ for classification trees. This should be short because you can reference the steps you wrote for the regression trees in (a).

For classification trees the steps are similar to regression trees with the following modifications to the corresponding steps...

2 and 3) Rather than minimizing weighted average SSE we use the Gini metric

4) Rather than using $\bar{y}$ we use mode(y)

(g) [difficult] Think of another objective function that makes sense besides the Gini that can be used to compare the “quality” of splits within inner nodes of a classification tree.

Something I have used as a feature in my personal research is entropy scoring. In order to adopt this to be used towards classification we can take 1-entropyScore as a metric of Information gained. We want to minimize entropy in each split so that our information gained in maximized. Essentially we can think of this process as wanting each side of our splits to be as pure (of one class) as possible.

Mathematically, we can define entropy as...

$E = -\sum_{i=1}^{n} P(x_i) log_2 P(x_i)$, where $P(x_i)$ is the probability of randomly choosing the ith element. Once we have done this for each split we can say that the information gain is $I = 1 - E$, where I is the "information" obtained. In information theory when entropy is high, we consider this to be more valuable as surprising results are considered informative. Here we are treating information gain a bit differently, more of a measure of informative purity. In our case the lower the entropy the better and the closer to a score of 1 we will get.

OR simply as we stated in class,

Misclassification error can be used.