The Gory Details

Pavel Komarov

March 12, 2018

Here lies a full explanation of how multivariate Projection Pursuit Regression (PPR) and univariate Classification work, to the degree I currently understand. It is as much for me as for you, because I much prefer my own notation, and every time I have to dig in to the bones of the code and faff with the loss function, I end up having to refer to the original paper's somewhat ambiguous, derivationless equations. Formerly I was attempting to deposit some of this knowledge in code comments, but they took up too much space while somehow remaining marvelously unreadable.

This is all in LaTeX because native math in markdown is amazingly still not supported by github, so a `.md` would really be no less uncomfortable than code comments. If you happen to want to know how I accomplished all this gorgeous formatting, the source `.tex` is in the /doc subdirectory of the repo and can be compiled with `pdflatex`.

I attempt to follow a structure based on the five components of any machine learning algorithm:

1. A Task/Problem Reduction
2. A Model
3. A Loss Function
4. An Optimization Scheme
5. Data

–or at least the middle three, since by the time the data for your task meets my algorithm you'll have abstracted away concerns about where it came from and what it actually means.

1 The Model

PPR is a statistical model of the form:

$$\vec{y_i} = \sum_{j=1}^{r} f_j(\vec{x_i} \cdot \vec{\alpha_j}) \otimes \vec{\beta_j}^T$$

where:

- i iterates examples, the rows of input and output matrices
- j iterates the number of terms in the PPR "additive model"
- r is the total number of projections and functions (terms) in the PPR
- $\vec{y_i}$ is a d-dimensional vector, the i^{th} row in an output matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times d}$
- $\vec{x_i}$ is a p-dimensional vector, the i^{th} row of an input matrix $\boldsymbol{X} \in \mathbb{R}^{n \times p}$
- $\vec{\alpha_j}$ is the j^{th} projection vector in the mdoel, a p-dimensional vector inner-producted with $\vec{x_i}$
- f_j is the j^{th} function in the model, mapping from $\mathbb{R}^1 \to \mathbb{R}^1$

- $\vec{\beta_j}^T$ is the transpose of $\vec{\beta_j}$, a d-dimensional vector outer-producted with the result of f_j to yield a result in the output space
- $\cdot$ is an inner product
- $\otimes$ is an outer product

This can be "vectorized" as

$$\hat{\boldsymbol{Y}} = \sum_{j=1}^{r} f_j(\boldsymbol{X} \cdot \vec{\alpha_j}) \otimes \vec{\beta_j}^T$$

where the "hat" $\hat{}$ distinguishes the answers predicted, as opposed to known training answers.

I also term this the "evaluation function". It may seem complicated, but the idea is simple:

1. Linearly project the input down to one dimension where it is easier to work with, thereby sidestepping the curse of dimensionality.
2. Find a sensible mapping from this reduced space to weighted "residuals", unexplained variance in the outputs. This is where the nonlinearity happens.
3. Unpack from the single-dimensional residual space to the output space with a kind of inverse projection.

1.1 A Word on Additive Models

In practice a single projection-mapping-expansion is not descriptive enough to capture the richness of what may be a very complicated underlying relationship between $\boldsymbol{X}$ and $\boldsymbol{Y}$, so it is repeated r times, each new "stage" only accounting for the variance left unexplained by the stages that have come before. Notice that, as per Taylor's Theorem and the no-doubt familiar universal approximation theorems, for certain classes of functions f, as r goes to infinity the evaluation function can approximate any continuous functional relationship between inputs and outputs.

2 The Loss Function

The (supervised) learning process consists of minimizing a standard quadratic cost, or "loss", function:

$$loss = \sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2$$

where:

- i iterates all training examples
- n is the total number of training examples
- w_i is the weight of the i^{th} example
- y_i is the known answer for example i
- $\hat{y}_i$ ("y-i-hat") is the answer predicted by the model for example i

In words: Get as close as you can for all examples, where the penalty for predicting an example incorrectly scales with the square of how wrong you are.

There are other choices of loss function, including an absolute value loss with no square penalty. This can work better in cases where a nonliner penalty might cause the algorithm to account too strongly for a few outlier examples.

(TODO) It is also possible to add regularization terms, which bias the optimization process to find smaller parameter values. This is good for numerical stability and has the added benefit of reducing the likelihood of overfit.

Unfortunately, modifying the loss function in either of these ways means altering the optimization process, and that's a mathematical bear I just haven't chosen to wrestle with.

Plugging the evaluation function in to the cost function yields a relationship between model parameters and loss. Because there are multiple dimensions in our vector $\vec{y_i}$, we introduce a sum over them so the PPR is motivated to make good predictions for all entries of the output:

$$loss = \sum_{i=1}^{n} w_i \sum_{k=1}^{d} w_k [y_{ik} - \sum_{j=1}^{r} f_j(\vec{x_i} \cdot \vec{\alpha_j}) \otimes \beta_{jk}]^2$$

where this new fauna:

- k iterates the columns of the output $\boldsymbol{Y}$
- d is the number of outputs, the width of the output matrix $\boldsymbol{Y}$
- w_k is a scalar weight, the relative importance of the k^{th} output dimension
- y_{ik} is the scalar k^{th} entry in the vector $\vec{y_i}$, itself the i^{th} row of $\boldsymbol{Y}$
- β_{jk} is the scalar k^{th} entry of $\vec{\beta_j}$ from the evaluation function

This can be vectorized as

$$loss = (\vec{w_I}^T \cdot (\boldsymbol{Y} - \hat{\boldsymbol{Y}})^2) \cdot \vec{w_K}$$

where

- $\vec{w_I}$ is an n-vector containing all example weights, w_i
- $\vec{w_K}$ is a d-vector containing all output dimension weights, w_k

and the sums disappear inside the inner products.

The parameters we need to optimize to make the PPR "learn" are $\vec{\alpha_j}$, f_j, and $\vec{\beta_j}$. w_k are hyperparameters chosen by the user, just as r is chosen. If output dimensions are on differing scales and the user wishes for the loss to be relative to those scales, then choosing w_k inversely proportional to the variance of outputs causes the loss to treat outputs more evenly. If some output dimensions are more important than others, then setting their weights higher causes predictions in those dimensions to be closer.

3 The Optimization Scheme

The macroscopic optimization scheme to solve for so many different parameters is non-obvious but straightforward:

1. Initialize all $\vec{\alpha_j}$, f_j and $\vec{\beta_j}$ to something random. Let $j = 1$.
2. Find the "residual" variance unexplained by all stages fit so far.
3. Project the input in to single dimension: $\boldsymbol{X} \cdot \vec{\alpha_j}$.

4. Fit f_j to a weighted residual target versus projections.
5. Use this f_j to find a better setting for $\vec{\beta_j}$.
6. Use a Gauss-Newton scheme to solve for an update to $\vec{\alpha_j}$.
7. Repeat steps 3-6 until f_j, $\vec{\beta_j}$, and $\vec{\alpha_j}$ converge.
8. (optional) Use the newly converged parameters to retune all previous f_t, $\vec{\beta_t}$, $\vec{\alpha_t}$ where $t \leq j$. (backfitting)
9. Increment j and go back to step 2 until j reaches r.

The inner loop here is a form of *alternating optimization*, wherein all parameters except one are held constant, the best setting for that parameter given those constants is found, and the process cycled through all parameters until convergence. The outer loop builds up the additive model.

But this leaves some details unexplained. How exactly is the residual found? How are parameters found given fixed solutions to the others? How do we test for convergence?

3.1 Finding The Residual

The residual trick cleverly separates the contribution of the j^{th} stage from the other terms in the additive model. Consider rephrasing the loss function as:

$$loss = \sum_{i=1}^{n} w_i \sum_{k=1}^{d} w_k [y_{ik} - \sum_{t \neq j} f_t(\vec{x_i} \cdot \vec{\alpha_t}) \otimes \beta_{tk} - f_j(\vec{x_i} \cdot \vec{\alpha_j}) \otimes \beta_{jk}]^2$$

Now if we let

$$r_{ijk} = y_{ik} - \sum_{t \neq j} f_t(\vec{x_i} \cdot \vec{\alpha_t}) \otimes \beta_{tk}$$

then

$$loss,\ L = \sum_{i=1}^{n} w_i \sum_{k=1}^{d} w_k [r_{ijk} - f_j(\vec{x_i} \cdot \vec{\alpha_j}) \otimes \beta_{jk}]^2$$

In practice we will wish to find all r_{ijk} for a particular j. Call this $\boldsymbol{R_j} \in \mathbb{R}^{n \times d}$, the same space as the output. It can be found with

$$\boldsymbol{R_j} = \boldsymbol{Y} - \sum_{t \neq j} f_t(\boldsymbol{X} \cdot \vec{\alpha_t}) \otimes \vec{\beta_t}$$

where $f_t(\boldsymbol{X} \cdot \vec{\alpha_t})$ is actually an n-vector.

3.2 Optimizing $\vec{\beta_j}$ Given $\vec{\alpha_j}$ and f_j

$$loss\ for\ the\ jth\ term,\ L_j = \sum_{i=1}^{n} w_i \sum_{k=1}^{d} w_k [r_{ijk} - f_j(\vec{x_i} \cdot \vec{\alpha_j}) \otimes \beta_{jk}]^2$$

To optimize with respect to a parameter, use good ol' calculus: Take a derivative, set equal to zero, and solve. Let's select $\beta_{jk'}$, the $(k = k')^{\text{th}}$ entry of $\vec{\beta_j}$ as the parameter of interest.

$$\frac{\partial L_j}{\partial \beta_{jk'}} = \sum_{i=1}^{n} w_i w_{k'} [2(r_{ijk'} - f_j(\vec{x_i} \cdot \vec{\alpha_j}) \otimes \beta_{jk'})(-f_j(\vec{x_i} \cdot \vec{\alpha_j}))] = 0$$

Notice that the sum over k disappears because no term where $k \neq k'$ will contain our variable $\beta_{jk'}$, so for the purposes of differentiation they are constant, and the derivative of constants is zero. Only the weight $w_{k'}$ remains.

Also, I've been using $\otimes$ for consistency, but since $\beta_{jk'}$ is scalar, an outer product is nothing special, just an ordinary multiplication. So we can do some algebra:

$$-2w_{k'}\sum_{i=1}^{n} w_i[r_{ijk'}f_j(\vec{x_i}\cdot\vec{\alpha_j})] + 2w_{k'}\beta_{jk'}\sum_{i=1}^{n} w_i[f_j^2(\vec{x_i}\cdot\vec{\alpha_j})] = 0$$

$$\rightarrow \beta_{jk'} = \frac{\sum_{i=1}^{n} w_i[r_{ijk'}f_j(\vec{x_i}\cdot\vec{\alpha_j})]}{\sum_{i=1}^{n} w_i[f_j^2(\vec{x_i}\cdot\vec{\alpha_j})]}$$

This can be vectorized to find all entries of $\vec{\beta_j}$ at once:

$$\vec{\beta_j} = \frac{\boldsymbol{R}_j^T \cdot (\vec{w_I} \odot f_j(\boldsymbol{X}\cdot\alpha_j))}{f_j(\boldsymbol{X}\cdot\alpha_j)\cdot(\vec{w_I}\odot f_j(\boldsymbol{X}\cdot\alpha_j))}$$

where $\odot$ is a Hadamard product, $\vec{w_I}$ is an n-vector, $\boldsymbol{R}_j$ is the residuals, and the dot products are taken along the length-n dimensions of these objects.

3.3 Optimizing f_j Given $\vec{\alpha_j}$ and $\vec{\beta_j}$

Now a similar argument, but this time consider the parameter of interest to be $f_{i'j}$, the $(i = i')^{\text{th}}$ entry of the n-vector formed by taking the inner product of $\boldsymbol{X}$ with α_j and applying f_j to each entry.

$$\frac{\partial L_j}{\partial f_{i'j}} = w_{i'}\sum_{k=1}^{d} w_k[2(r_{i'jk} - f_{i'j}\otimes\beta_{jk})(-\beta_{jk})] = 0$$

The sum over i disappears because only the single term where $i = i'$ isn't constant to the derivative.

$$\rightarrow w_{i'}\sum_{k=1}^{d} w_k[-2r_{i'jk}\beta_{jk} + 2f_{i'j}\beta_{jk}^2] = 0$$

$$\rightarrow 2f_{i'j}\sum_{k=1}^{d} w_k\beta_{jk}^2 = 2\sum_{k=1}^{d} w_k r_{i'jk}\beta_{jk}$$

$$\rightarrow f_{i'j} = \frac{\sum_{k=1}^{d} w_k r_{i'jk}\beta_{jk}}{\sum_{k=1}^{d} w_k\beta_{jk}^2}$$

As in the case of $\vec{\beta_j}$, this can be vectorized.

$$f_j(\boldsymbol{X}\cdot\vec{\alpha_j}) = \frac{\boldsymbol{R}_j\cdot(\vec{w_K}\odot\vec{\beta_j})}{\vec{\beta_j}\cdot(\vec{w_K}\odot\vec{\beta_j})}$$

This provides targets for the funtion f_j. The task is to find the function that maps from this input to this output, for which there are numerous solvers (finding a polynomial by reducing to a linear inverse problem, for example). The example weights w_i disappear in the algebra and so do not affect the targets, but they can be passed on to the function-fitter so it considers some examples more important than others.

3.4 Optimizing $\vec{\alpha_j}$ Given $\vec{\beta_j}$ and f_j

This is by far the toughest set of parameters to optimize, because they are nested inside the function. This time express the loss as:

$$L_j = \sum_{k=1}^{d} w_k [\vec{w_I} \odot \vec{g_{jk}}^2]$$

where

$$\vec{g_{jk}} = \vec{r_{jk}} - f_j(\boldsymbol{X} \cdot \vec{\alpha_j}) \otimes \beta_{jk}$$

where $\vec{r_{jk}}$ is the n-vector formed by stacking $r_{ijk} \forall i$ together, or equivalently the k^{th} column of $\boldsymbol{R}_j$. The weights can be factored in to the square to yield a form solveable with Gauss-Newton.

$$L_j = \sum_{k=1}^{d} \vec{g_{jkw}} (\vec{\alpha_j})^2,$$

$$\vec{g_{jkw}} = \sqrt{w_k}\sqrt{\vec{w_I}} \odot (\vec{r_{jk}} - f_j(\boldsymbol{X} \cdot \vec{\alpha_j}) \otimes \beta_{jk})$$

Find the Jacobian:

$$J_{jk}[u, v] = \frac{\partial \vec{g_{jkw}}(\vec{\alpha_j})[u]}{\partial \vec{\alpha_j}[v]} = -\sqrt{w_k}\sqrt{w_u}\dot{f}_j(\vec{x_u} \cdot \vec{\alpha_j})\beta_{jk}x_{uv}$$

where w_u is $\vec{w_I}[u]$, the u^{th} example weight, as distinct from w_k, which, remember, are output dimension weights.

$$\rightarrow J_{jk} = -\sqrt{w_k}\beta_{jk} \begin{bmatrix} \sqrt{w_0}\dot{f}_j(\vec{x_0} \cdot \vec{\alpha_j})x_{00} & \sqrt{w_0}\dot{f}_j(\vec{x_0} \cdot \vec{\alpha_j})x_{01} & \dots & \sqrt{w_0}\dot{f}_j(\vec{x_0} \cdot \vec{\alpha_j})x_{0p} \\ \sqrt{w_1}\dot{f}_j(\vec{x_1} \cdot \vec{\alpha_j})x_{10} & \sqrt{w_1}\dot{f}_j(\vec{x_1} \cdot \vec{\alpha_j})x_{11} & \dots & \sqrt{w_1}\dot{f}_j(\vec{x_1} \cdot \vec{\alpha_j})x_{1p} \\ \vdots & & \ddots & \vdots \\ \sqrt{w_n}\dot{f}_j(\vec{x_n} \cdot \vec{\alpha_j})x_{n0} & \dots & \dots & \sqrt{w_n}\dot{f}_j(\vec{x_n} \cdot \vec{\alpha_j})x_{np} \end{bmatrix}$$

$$= -\sqrt{w_k}\beta_{jk}(\sqrt{\vec{w_I}} \odot \dot{f}_j(\boldsymbol{X} \cdot \vec{\alpha_j})) \odot \boldsymbol{X}$$

where everything left of the last $\odot$ simplifies to an n-vector, so the Hadamard product is taken with each column of $\boldsymbol{X}$ individually.

As per Gauss-Newton, the update to the parameter $\vec{\alpha_j}$ to the function $\vec{g_{jkw}}$ is given by the solution $\vec{\delta}$ to:

$$\left[\sum_{k=1}^{d} J_{jk}^T J_{jk}\right] \vec{\delta} = \sum_{k=1}^{d} J_{jk}^T \vec{g_{jkw}}$$

On the left side is a $p \times p$ matrix, and on the right a $p \times 1$ vector, so we have an easy-to-solve linear inverse problem.

$$\vec{\alpha_j} = \vec{\alpha_j} + \vec{\delta}$$

In practice $\vec{\alpha_j}$ is renormalized after this update to avoid numerical drift and keep it from blowing up. Because $\vec{\alpha_j}$ is a projection vector only meant to flatten the data along some dimension, its magnitude does not really matter.

3.5 Testing for Convergence

Parameters are considered "converged" when their values across iterations cause a small-enough change in the value of the loss function.

Because we only make updates to the parameters governing stage j, we can make use the convenient identity

$$\boldsymbol{Y} - \hat{\boldsymbol{Y}} = \boldsymbol{R}_j - f_j(\boldsymbol{X} \cdot \vec{\alpha_j}) \otimes \vec{\beta_j}^T$$

to calculate the loss quickly.

4 Classification

Let the miscalssification risk R be

$$R = \sum_{i=1}^{n} \min_{k \in [1,q]} \sum_{c=1}^{q} l_{ck} p(c|\vec{x_i})$$

where

- i iterates over examples
- q is the total number of classes in the problem
- $\min_k$ implements the optimal decision rule for each example
- l_{ck} is the user-specified loss for predicting $y = k$ when in truth $y = c$
- the inner sum is the total loss for predicting $y = k$
- $p(c|\vec{x_i})$ is the true probability $y = c$ given input $\vec{x_i}$

The unknown here is that conditional probability. If we define an indicator variable

$$h_{ci} = 1 \; if \; y_i = c, \; 0 \; otherwise$$

then Friedman says we can rewrite the conditional probability as

$$p(c|\vec{x_i}) = \frac{\pi_c S}{s_c} E[h_{ci}|\vec{x_i}]$$

where

- π_c is the prior probability that $y_i = c$ ($h_{ci} = 1$), calculable from the training set with $\frac{\sum_{i=1}^{n} h_{ci}}{n}$
- $s_c = \sum_{i=1}^{n} w_i h_{ci}$, the cumulative weight of examples with classification c
- $S = \sum_{c=1}^{q} s_c$, the cumulative weight of everything, a constant
- E means the expected value

The coefficient can be expanded to

$$\frac{\sum_{c=1}^{q} \sum_{i=1}^{n} w_i h_{ci} * \frac{\sum_{i=1}^{n} h_{ci}}{n}}{\sum_{i=1}^{n} w_i h_{ci}} \propto \frac{\sum_{i=1}^{n} h_{ci}}{\sum_{i=1}^{n} w_i h_{ci}}$$

This doesn't make any sense, because it says probabilities are *inversely* related to example weights, but we really want the probability of predicting a given class to be *directly* related to the importances of examples in that class. Convinced Friedman accidentally inverted the equation, the code and the following equations reinvert his inversion.

Notice that if the weights w_i in s_c are uniform (so no example is considered any more important than any other), then $\pi_c = s_c/S$, and all those terms cancel (regardless of the inversion).

Additionally, l_{ck} is often simplified as

$$l_{ck} = 1 \; if \; c \neq k, \; 0 \; if \; c = k$$

So rewrite the risk as

$$R = \sum_{i=1}^{n} \min_{k \in [1,q]} \frac{1}{S} \sum_{c=1}^{q} \frac{s_c l_{ck}}{\pi_c} E[h_{ci} | \vec{x_i}]$$

And with the simplifying assumptions that all examples are equally important and misclassification is equally bad between all class pairs it becomes:

$$R = \sum_{i=1}^{n} \min_{k \in [1,q]} \sum_{c \neq k} E[h_{ci} | \vec{x_i}]$$

Or equivalently:

$$R = \sum_{i=1}^{n} \max_{k \in [1,q]} E[h_{ci} | \vec{x_i}]$$

because the sum is minimized by excluding the largest expectation.

Now, recognize $E[\vec{h_c}|\boldsymbol{X}]$ is a vector of the values $E[h_{ci}|\vec{x_i}]$, and for training data the expectation that h_{ci} has a given value given $\vec{x_i}$ is known to be either a one or a zero, so $E[\vec{h_c}|\boldsymbol{X}] = \vec{h_c}, \; h_{ci} \forall i$ stacked together.

Further, recognize that stacking $\vec{h_c}$ for all classes c together as columns yields $\boldsymbol{H}$, a one-hot representation of the true classifications $\boldsymbol{Y}$. That is:

$$\boldsymbol{Y} = \begin{bmatrix} 1 \\ 0 \\ 2 \\ . \\ . \\ 3 \end{bmatrix} \qquad \boldsymbol{H} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ & \dots & \dots & \\ & \dots & \dots & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And now we can model $\boldsymbol{H}$ with a multivariate projection pursuit regression model, where we take the predicted class of example i to be:

$$\hat{y_i} = \underset{c}{\operatorname{argmax}} \, \hat{h_{ci}}$$

That is: the predicted class is the index of the column where the largest value in the i^{th} row of $\hat{\boldsymbol{H}}$, the predicted $\boldsymbol{H}$, is located. If $\boldsymbol{Y}$ is filled with generalized categories rather than numbers, then categoricals can be assigned numbers for the construction of $\boldsymbol{H}$, and argmaxes can be translated back at prediction-time.

And just like that we have reduced univariate classification to multivariate regression! There is one caveat: Training the model to make these predictions should ideally involve optimizing the the misclassification risk as the loss function, not the sum-of-squares loss function from section 2 as is done for actual regression. But the $\max_k$ in the risk equation makes it nonconvex, which means we can no longer employ the methods detailed in section 3 to find model parameters.

Fortunately, Friedman assures us (and experiment bears out) that using the quatratic loss function is acceptable, and if we wish to account for examples being of differing importances or specify a funky non-uniform pairwise loss scheme, all we have to do is use weights:

$$w_c = \frac{s_c}{S\pi_c} \sum_{k \in [1,q]} l_{ck}$$

Since weights are relative, the constant S and the normalization by the constant n in the calculation of π_c can be dropped to yeild:

$$w_c = \frac{\sum_{i=1}^{n} w_i h_{ci}}{\sum_{i=1}^{n} h_{ci}} \sum_{k \in [1,q]} l_{ck}$$

Vectorizing to obtain this weight for all classes at once we get:

$$\vec{w_C} = \frac{\vec{w_I} \cdot \boldsymbol{H}}{\sum_{i=1}^{n} \boldsymbol{H}} \odot \sum_{k=1}^{q} \boldsymbol{L}$$

where $\boldsymbol{L}[c,k] = l_{ck}$, the sum over k amounts to a sum along the horizontal (second) axis, the sum over n of $\boldsymbol{H}$ amounts to a sum along the vertical (first) axis, and the division is pointwise on the vectors involved.