Chapter 4 - Interpretable Models

1 Introduction

Let's talk about how to interpret some specific models. Monotonicity means that an increase in feature value always has a single impact on prediction (RuleFit, k nearest neighbors, and (sort of) decision trees are non-monotonic). RuleFit and decision trees are the only models that can also account for interactions between features.

2 Linear Regression

The prediction is $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_p x_p$ where the weights $\hat{\beta}_0, ..., \hat{\beta}_p$ are learned by minimizing squared error. You can construct confidence intervals around the weights to measure uncertainty.

The linear model is a bad fit if your data is not linear, if noise is not normally distributed, if noise does not have constant variance, if instances are not independent of each other, or features are strongly correlated.

Increasing a numerical feature increases the prediction by its weight. For a binary feature, changing from reference category (0) to the other category increases the prediction by its weight. For a categorical feature with multiple categories, you can use one-hot encoding (with $L-1$ entries for L categories - all 0 is the reference category). The intercept β_0 is the prediction when all categorical features are at the reference class and all features are at 0 (so make sure to give numerical features zero mean and unit variance).

To measure how well your model fits the data, use $R^2 = 1 - SSE/SST$ where $SSE = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$ and $SST = \sum_{i=1}^{n} (y^{(i)} - \bar{y})^2$. Actually, you should use $\bar{R}^2 = R^2 - (1 - R^2)\frac{p}{n-p-1}$ where n and p are the number of instances and features, respectively. R^2 is between 0 and 1, where larger values indicate better fit.

The importance of a feature is $t_{\hat{\beta}_j} = \frac{\hat{\beta}_j}{SE(\hat{\beta}_j)}$.

One visual interpretation is a weight plot, make the feature name the y axis. Make the feature weight the x axis, and plot the feature weight (and confidence interval) for each feature. In order to compare features, make sure they all have zero mean and unit variance.

Another visual is the effect plot. $effect_j^{(i)} = w_j x_j^{(i)}$. Now make the y axis the feature name, x axis the effect, and plot a box and whiskers plot for each feature. Again, features should have zero mean and unit variance. This visualization handles categorical variables better than the weight plot. If you are considering an individual prediction, you can compute its effects and overlay them on the corresponding box and whiskers plots.

Suppose we have six examples that take on categories A, A, B, B, C and C, respectively. How do we encode this? In treatment coding, A is the reference category $1, 0, 0$, B is $1, 1, 0$, and C is $1, 0, 1$. Notice that the first 1 is fixed. In effect coding, we get A $1, -1, -1$, B $1, 1, 0$, and C $1, 0, 1$. In dummy coding we get A $1, 0, 0$, B $0, 1, 0$, and C $0, 0, 1$.

Linear models are contrastive, not selective, simple, and general. If your data is linear, they are truthful.

If you want to have a model that gives 0 weight to less important features, add the L1 regularization term $\lambda||\beta||_1$ to the least squares objective. Pick lambda by putting feature weight on the y axis, $\log \lambda$ on the x axis, and plotting each feature weight as you vary lambda. Pick the lambda that gives you the desired number of nonzero features. Other options to get sparsity is to use domain knowledge to pick promising features, pick features correlated with the target, use forward selection (greedily add features that improve R^2 the most), or use backward selection.

Linear models are simple, well studied, available everywhere, and efficient. However, data is rarely linear so they don't perform well. They also struggle when you have strongly correlated features.

3 Logistic Regression

Linear regression does not work well for classification because it does not output probabilities, extrapolates, has no meaningful threshold value between classes, and does not generalize to multiple classes. Logistic regression overcomes these problems. For binary classification, logistic regression computes the log odds with $\log \frac{P(y=1)}{P(y=0)} = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$. You may find it easier to look at the odds instead of the log odds. You can also solve for $P(y = 1)$ by noting that $P(y = 0) = 1 - P(y = 1)$.

Logistic regression has the same pros and cons of linear regression. It is more complicated than the linear regression model because of the log odds. It is nice because it gives you a probability. You can generalize it to multiclass classification (it is called multinomial classification in this case).

4 GLM, GAM and more

Let's look at ways to extend the linear regression model. Linear regression assumes the target is a linear function of the features plus Gaussian noise. If the noise is not Gaussian (e.g. the target cannot be negative), use a generalized linear model (GLM). If the features interact, add interactions manually. If the relationship is not linear, use a generalized additive model (GAM) or add feature transformations as features.

A GLM has $g(E_Y(y|x)) = \beta_0 + \beta_1 x_1 + ... + \beta_p x_p$, where g is the link function and Y is a probabiblity distribution from the exponential family. If the target is a count, use a Poisson distribution. If the target is always positive, use the exponential distribution, The linear regression model uses the Gaussian distribution. The logistic regression (multinomial regression) model uses a Bernoulli (multinoulli) distribution. Each distribution has a canonical link function (Gaussian has identity, Bernoulli has ln, Poisson has ln), but you can pick your own.

If features interact, add interactions as features. For example, multiply two numerical features. Or multiply numerical feature by each entry of one-hot encoded of categorical feature. Or compute all possible combinations of two categorical features and create a separate category for each and encode that. This can yield a lot of interaction features if applied naively, so use an algorithm like RuleFit to pick only the most promising interaction terms.

What if the underlying relationship between features and targets is not linear? One option is to transform a feature (e.g. take its logarithm, square root, or exponential) and add that as a new feature. Another option is to discretize numerical features and make them into a categorical feature (this is hard to do well though). Another option is to use a GAM, which models $g(E_Y(y|x)) = \beta_0 + f_1(x_1) + ... + f_p(x_p)$. A good choice for the f_j functions are splines, which are sums of curves (read about these elsewhere).

All these extensions to linear models can make them harder to interpret. There are also tons of other variants of linear models (data not i.i.d = mixed models or generalized estimating equations, noise is not heteroscedastic = robust regression, outliers in data = robust regression, predicting time until event occurs = parametric survival models, cox regression, survival analysis, predicting a category = multinomial regression or logistic regression, predicting ordered categories = proportional odds model, predicting a count = Poisson regression, predicting a count where 0 is common = zero-inflated Poisson

regression, hurdle model, not sure what features you need = causal inference, mediation analysis, missing data = multiple imputation, have prior knowledge = Bayesian inference).

5 Decision Tree

Decision trees split based on feature values until they hit a leaf node, which is where they make a prediction. CART is a popular algorithm for learning decision trees. The model is $\hat{y} = \hat{f}(x) = \sum_{m=1}^{M} c_m I[x \in R_m]$ where the leaf value (average of training examples in leaf) is c_m and $I[x \in R_m]$ indicates whether x follows the path to leaf m. CART greedily picks features and split points (or split subsets in the case of categorical features) to minimize some measure of heterogenaity in the leaves (variance for regression, Gini index for classification). The algorithm continues until hitting a maximum number of nodes or tree depth.

A decision tree is simply a flowchart, so it's easy to see how a prediction is made.

To compute feature importance, look at all splits where feature is used and sum up how much it reduces variance (or Gini index) - normalize importances so that their sum is 100.

The predicted value of an internal node is the average of all instances in the leaves under it. So, you can see how the predicted value is changing as you go through the tree. You can also compute the amount that each feature has contributed.

Decision trees are easily understood and can handle feature interactions. It also makes counterfactual analysis easy (what if I went down this path instead?). They also don't require you to standardize numerical features or encode categorical features. Their key disadvantages are that they can exactly model linear relationships, are not smooth, and are unstable (changing the dataset may yield very different tree).

6 Decision Rules

These are IF-THEN rules (IF = condition/antecedent, THEN = prediction). These rules can have ANDs in them. A rule is evaluated by its support (fraction of examples for which the antecedent applies) and accuracy (when the antecedent applies, what fraction of the time is it correct?). When you have many rules, some rules might overlap and there may be examples that do not fit any rules. We fix overlap with a decision list (i.e. we consider rules one at a time) or decision set (i.e. the rules run independently and we average their votes). We also make a default rule that applies if no other rule matches. We can learn rules with OneR, sequential covering, and Bayesian Rule Lists.

OneR first discretizes continuous features. Then for each feature we consider each discrete feature value, look at the dominant target value for that feature value (if you are doing regression, discretize the target), and create a rule and compute the error of the rule. We then pick the feature with smallest overall error. If features have many possible discrete values, OneR can overfit, so use a training set to make rules and validation set to evaluate error.

Sequential covering makes a decision list (or set). We iteratively build up the list (or set) by adding the rule to the list (or set) and removing points that the rule covers. We keep doing this until our rules meet some quality threshold. For multiclass settings, we treat the least common class as class 1 and all other classes as class 0. Once we finish learning class 1, we consider the next least common class and repeat the process. To learn a rule, we fit a decision tree and recursively select the purest nodes and turn the path from root to leaf into a rule. There are other ways to learn rules. RIPPER is a more sophisticated variant of sequential covering.

In Bayesian Rule Lists (BRL), we first pre-mine frequent patterns and then build a rule list from them. To mine frequent patterns, we don't need the targets. A pattern is just a set of (feature, feature value) pairs. The frequency is the support. You can mine frequent patterns with the Apriori or FP-Growth algorithms. Apriori finds frequent patterns and builds association rules, but we only care about finding

the frequent patterns. Apriori first finds single (feature, value) pairs whose support exceeds a threshold. It then combines these patterns with AND to look for higher order patterns with sufficiently high support. With these in hand, we can use BRL to incorporate prior knowledge that prefers short rules and a short list. We first generate an initial list from the prior distribution, then we sample lists by adding/removing rules, and finally we select the list with highest posterior probability. See the book for details.

Decision lists are very easy to interpret, fast to run, and robust against outliers. They also tend to produce sparse models, which is good. However, they are not great for regression or continuous features because you need to discretize. Except for RIPPER and BRL, they tend to overfit.

7 RuleFit

This learns sparse linear models that learns interaction effects (as decision rules). Basically, we learn decision trees, pick paths from root to leaf as decision rules, and use these as features along with the regular features, and do Lasso (L1) linear regression. You can interpret them with the same techniques we use for linear models. The importance of a feature should also include all the rules that it appears in.

Note that you can fit a random forest or gradient boosted tree on the data and then extract the rules from the resulting model. Then, take your features and windsorize them (i.e. constrain them between the 5th percentile and 95th percentile values). Next, add linear terms (one per windsorized feature) to the set of decision rules. These linear terms should be $0.4 * f/std(f)$ where f is the feature - we do this so they are comparable to the decision rule features. Then, do Lasso (L1) regression.

To compute feature importance for the linear terms, we do $I_j = |\beta_j| std(l_j(x_j))$ where β_j is the learned weight and $l_j(x_j)$ is the linear term. For decision rules we have $I_k = |\alpha_k|\sqrt{s_k(1-s_k)}$ where α_k is the learned weight and s_k is the support of the rule. The overall importance for a feature is the importance of its linear term plus its importance in all the rules that it appears in (a rule splits its importance evenly over the features that comprise it).

RuleFit is great because it mixes linear models and rule models. It also has good local interpretability because most rules will not apply for a given data point. It works well with methods like feature importance, partial dependence plots, and feature interactions. RuleFit can be confusing when there are too many rules or when there are overlapping rules.

8 Other Interpretable Models

Naive Bayes assumes features are independent and computes $P(C_k|x) = \frac{1}{Z} P(C_k) \sum_{j=1}^{p} P(x_j|C_k)$ where Z is a normalization constant that ensures the probabilities sum to 1. This is interpretable because you can see how much probability each feature contributes.

The k nearest neighbors method classifies a point by averaging the predictions for its k nearest neighbors in the training set. It is kind of interpretable at the local level because you can look at the neighbors.