Chapter 1 - Introduction

1 Introduction

This book describes how to make supervised machine learning models interpretable. Basically, you can use interpretable models or model-agnostic methods (can give local or global explanations).

2 Story Time

Three fictional stories that are supposed to motivate why interpretability is important. Skip.

3 What is Machine Learning?

We focus on supervised learning (not unsupervised or reinforcement learning). Machine learning often beats hand designed algorithms, but is hard to debug.

4 Terminology

An algorithm is a set of rules. Machine learning is a set of techniques for learning from data and making predictions. A machine learning algorithm defines how to create a machine learning model from data. A machine learning model maps inputs to outputs. A black box (in contrast to white box) model is one whose internal mechanisms are not decipherable (e.g. a complicated neural net is a black box model, a simple linear regression is white box).

Interpretable machine learning is a set of techniques that aims to make a model's predictions understandable to humans.

A dataset is made of (features, target) pairs. We assume features are interpretable. A prediction is a model's estimate of the target.

Chapter 2 - Interpretability

1 Introduction

Interpretability (or explainability) is the degree to which a human can understand why a model made a particular decision. Alternatively, it is the degree to which a human can predict a model's result.

2 Importance of Interpretability

Interpretability helps us understand why a model made a particular prediction. This is useful in debugging the model, gleaning insights into how the model works (and thus how it can be improved), discovering model weaknesses, complying with regulations, removing prejudice from models, and providing explanations to customers.

Models that explain their predictions may be more socially acceptable and offer a better user experience than black box models.

Interpretability allows us to examine the following properties of the model: fairness (model should not be prejudiced), privacy (model is not exposing or using sensitive information), reliability/robustness (small changes in input should not cause big changes in output), and causality (model uses causal features for prediction).

We don't need interpretability for hobby projects, for well understood problems (e.g. optical character recognition), or when malicious actors might try to game the model (e.g. a security model that explains how it detects bad actors might give them hints on how to circumvent it).

3 Taxonomy of Interpretability Methods

Interpretability can be intrinsic (the model is simple, like a decision tree) or post-hoc (i.e. we use some algorithm to extract interpretations). Here are the possible outputs of an interpretation method.

A feature summary statistic is a number about a feature or feature pair. A feature summary visualization (e.g. partial dependence plot) shows the interpretation visually. Model internals like the splits of a decision tree or visualization of the filters learned in a convolutional neural net can also provide interpretability. The output could be a data point, as in class prototypes or the counterfactual method. Finally, the output could be an intrinsically interpretable model that approximates the original black box model.

Interpretation tools can be model specific or model agnostic. They can explain an individual prediction, a local group of predictions, or the global set of all predictions.

4 Scope of Interpretability

Algorithm transparency is about how a particular learning algorithm learns from data. For example, we understand how least squares works and roughly how a convolutional neural net works (i.e. successive

layers learn edges, shapes, parts, and objects). We don't understand how complicated neural nets or gradient boosted trees learn though.

Global, holistic model interpretability is about how a trained model makes decisions in terms of its features and weights. Which features are important? How do they interact? This is basically impossible to do if you have more than 2 or 3 features.

Global model interpretability on a modular level is about understanding a model by looking at its pieces (e.g. one decision tree in a random forest). Note that a linear model's weights cannot be interpreted by considering just one weight, you need to consider multiple features and their weights.

Local interpretability for a single prediction is about understanding why a particular prediction was made and what features were important. LIME works well here.

Local interpretability for a group of predictions is about understanding a group of predictions. You can either run a global method on the group or a local method on each prediction and aggregate.

5 Evaluation of Interpretability

One option is application level evaluation. Just deploy the method and see if users indicate that they like it. Alternatively, you can have some experts try to concoct explanations and see if your method produces similar explanations.

Another option is human level evaluation. Get some volunteers, show them a bunch of examples each with different possible explanations, and ask them to pick the best ones.

A third option is function level evaluation. If your explanations take the form of a decision tree, you could ensure that the decision trees are not too deep.

6 Properties of Explanations

Explanations can come in many forms. They may be text or a visualization that ties feature values to the prediction. They may be a set of data points (e.g. in k nearest neighbors). Or they may take the form of simple, intrinsically interpretable models like decision trees.

The important properties of explanation methods are expressive power (how rich is the "language" used to generate explanations?), translucency (how much does the method depend on looking inside the model?), portability (how many different kinds of models does the explanation method work with?), and algorithmic complexity (how much compute time is needed to create the explanation?).

For individual explanations, we have some important properties. Accuracy is how well the explanation predicts unseen data. If the explanation, is "The house was priced as expensive because it has many rooms", then this should be true of other homes too. Fidelity is how well the explanation reflects what is actually going on in the model. Fidelity can be local or global. Consistency is how similar explanations are between two different models trained for the same task. Note that if two models arrive at the same result using different features, it may actually be possible that both are correct and could have different explanations (Rashomon Effect). Stability is how similar explanations are for similar examples (i.e. slight changes in feature values should not change explanation a lot). Comprehensibility is how well humans understand the explanations (e.g. is the explanation short? can people predict what the explanation will be for a new data point?). Certainty is whether the explanation indicates the confidence level of the model (assuming the model outputs a score instead of a label). The degree of importance is whether the explanation shows how important each part of the explanation is. Novelty is whether the data point being considered is very different from typical data points. Representativeness is how many examples the explanation covers (e.g. a global explanation covers all of them).

7 Human-friendly Explanations

A good explanation has just one or two causes and contrasts the current situation with one where the prediction would have been different.

An explanation answers a "why" question.

Explanations are contrastive, so they should explain why a different prediction was not made (e.g. why was this loan application not accepted?). It should explain the difference between the current example and some reference example.

Explanations are selective, so pick out the one or two key causes.

Explanations are social, so make sure your audience will understand them.

Explanations focus on the abnormal, so point out what makes the current data point unique. If a feature value is atypical, point that out.

Explanations are truthful, so make sure that it reflects what is going on in the model.

Explanations are consistent with user's beliefs. For example, people know that larger houses are more expensive, so your explanation should not contradict that.

Explanations are general and probable, so they should explain other similar situations (unless the current data point is actually an abnormal case). You can measure this by feature support, which is the fraction of instances to which the explanation applies.

Chapter 3 - Datasets

1 Introduction

The bike rental dataset is a regression problem where we use day of week, time, user numbers, and weather to predict how many bikes will be rented.

The YouTube spam dataset is a binary classification problem where we classify a comment as spam or not.

The cervical history dataset is a binary classification dataset that uses medical history features to predict whether a patient will get cervical cancer.

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.

Chapter 5 - Model-Agnostic Methods

1 Introduction

Let's look at methods that work for all kinds of models. If you do not use these, you should use an interpretable model or an interpretability method designed for the specific kind of model that you are using.

2 Partial Dependence Plot (PDP)

A PDP aims to plot the average model prediction as a function of feature value, for some feature of interest. On the x axis, we vary our feature of interest. On the y axis, we need to compute the average model prediction. To do this for a chosen value of our feature of interest, we simply consider each training example, plug in the chosen value of our feature of interest while keeping all other features untouched, compute the model prediction for this modified training example, and average the predictions for all the modified training examples.

The formula is as follows, where x_S is the value for the feature of interest, and $x_C^{(i)}$ are the values for the other features for the ith example.

$$\hat{f}_{x_S}(x_S) = \frac{1}{n}\sum_{i=1}^{n} \hat{f}(x_S, x_C^{(i)}) \tag{1}$$

When showing a PDP, it helps to show a feature histogram below them because users can then see how frequent a particular feature value is in the dataset.

PDPs have two limitations. The first is that they assume the feature of interest is uncorrelated to all other features. To overcome this limitation, use ALE plots. The second limitation is that we may have heterogeneity in feature effect. Heterogeneity in feature effect means that for some examples, increasing the feature value increases the model prediction while for other examples, increasing the feature value decreases the model prediction. These effects might cancel out in a PDP and the user will not know that they exist. To overcome this limitation, use ICE plots.

3 Individual Conditional Expectation (ICE)

Again, pick a feature of interest, vary it and compute the model prediction for each modified training example. However, instead of averaging the predictions, plot a separate prediction vs. feature value curve for each training example. This will make heterogeneity in feature effect clearly visible to the user. You may also want to plot the derivative of each curve to make it clearer. Also, you can make it so that all the curves start at the same value (an anchor point).

ICE plots overcome the heterogenaity issue of PDP plots, but they still suffer from the correlation issue.

4 Accumulated Local Effects (ALE) Plot

Let's understand the correlation effect. Suppose we are trying to predict a person's age given measurements like their height, weight, and other factors. If height is our feature of interest, a PDP will vary height without changing weight, so we may get ridiculous modified training examples like a person who is 6 feet tall but weighs 30 pounds. This happens because PDP does not know that height and weight are correlated.

An M-plot can partially overcome this by only considering training examples where the height is similar to the chosen value of height. So, for height = 3 feet, we consider examples where height is near 3 feet (e.g. between 2.5 feet and 3.5 feet). We then modify them to have 3 feet height, compute the model prediction, and average over the predictions. The problem with M-plots is that our model might not use the height feature at all, instead relying only on the correlated weight feature, but since we are implicitly using height when we select training examples, the M-plot incorrectly shows that changing height does impact model prediction.

ALE plots overcome the limitation of M plots by computing differences in model predictions rather than using model predictions directly. The equation is:

$$\hat{\tilde{f}}_{j,ALE}(x) = \sum_{k=1}^{k_j(x)} \frac{1}{n_j(k)} \sum_{i:x_j^{(i)} \in N_j(k)} \hat{f}(z_{k,j}, x_{-j}^{(i)}) - \hat{f}(z_{k-1,j}, x_{-j}^{(i)}) \tag{2}$$

Basically, we take the domain for the feature of interest and break it into intervals. Then, for each example in an interval, we plug in the upper value of the interval (leave the other features untouched), compute the model prediction, repeat this with the lower value of the interval, and take the difference between the two predictions for each example. This difference is called the local effect. We then average the effects over all the examples.

If we want, we can also subtract the mean effect. So, the ALE is the main effect of the feature at a certain value compared to the average prediction of the data.

$$\hat{f}_{j,ALE}(x) = \hat{\tilde{f}}_{j,ALE}(x) - \frac{1}{n} \sum_{i=1}^{n} \hat{\tilde{f}}_{j,ALE}(x_j^{(i)}) \tag{3}$$

How do we pick the intervals? A common choice is to just use quantiles.

We can extend ALE plots to 2 dimensions as well, but this shows second order effects rather than first order effects. The equations are a lot uglier in this case too.

How do we handle categorical features? They do not have an order so we cannot make intervals. To impose an order, we can compute the similarity of two categories by measuring how similar the other features are. See the book for how to do this.

To look for correlated features, we can use ANOVA. Basically, we turn one feature into the target and try to predict it from the other features. We see which features are the most useful for predicting it.

ALE plots are efficient, interpretable, and work even when features are correlated. They can suffer if you don't use a good number of intervals and they are hard to implement.

Use ALE instead of PDP.

5 Feature Interaction

Sometimes, the interaction of two features is what really matters. The H statistic can help here. It can help us tell if two given features interact or whether a given feature interacts with some subset of the

rest of the features.

For a pair of features that do not interact, we can use their partial dependence (see the PDP section) functions to get the following decomposition:

$$PD_{jk}(x_j, x_k) = PD_j(x_j) + PD_k(x_k) \tag{4}$$

If a given feature does not interact with the rest of the features, we have

$$\hat{f}(x) = PD_j(x_j) + PD_{-j}(x_{-j}) \tag{5}$$

By observing how much the actual partial dependence differs from this non-interacting case, we can get a measure of interaction strength called the H statistic. Here are the H statistics for the two feature and single feature cases described before:

$$H_{jk}^2 = \sum_{i=1}^{n} [PD_{jk}(x_j^{(i)}, x_k^{(i)}) - PD_j(x_j^{(i)}) - PD_k(x_k^{(i)})]^2 / \sum_{i=1}^{n} PD_{jk}^2(x_j^{(i)}, x_k^{(i)}) \tag{6}$$

$$H_j^2 = \sum_{i=1}^{n} [\hat{f}(x^{(i)}) - PD_j(x_j^{(i)}) - PD_{-j}(x_{-j}^{(i)})]^2 / \sum_{i=1}^{n} \hat{f}^2(x^{(i)}) \tag{7}$$

Notice that the H statistic takes $O(n^2)$ time to run. To speed it up, you can sample from the training set. The statistic varies between 0 and 1 where 0 means no interaction. There's also a hypothesis test to check if there is interaction, but your model needs to somehow make sure there is no interaction between any features (null hypothesis), which may not be possible for all models. Finally, you can also use the H statistic in classification when the output is a probability.

To use the H statistic in practice, first compute the single feature statistic for all features. Then, for highly interacting features, compute all the pairwise statistics involving the feature of interest. You can also plot 2D partial dependence plots here.

The H statistic suffers for strongly correlated features, does not work on image pixel features, and it can be hard to tell whether the statistic is large or small.

6 Feature Importance

First, compute the total error of the model on the training examples. Next, for your feature of interest, permute the feature values in the training examples. Compute the total error again. Compute the difference (or ratio) between the two errors. The larger the error, the more important the feature is to the model. This is called permutation feature importance.

With the above approach, you can compute the feature importance of all your features. You could also use the testing examples instead of the training examples (it's not clear whether this is better or not).

For your error function, use $1 - AUC$ for classification and mean absolute error for regression.

Using the ratio instead of difference makes the importance comparable accross different problems.

By the way, another way to compute feature importance is to just remove the feature and look at how error changes. The error change will not be as substantial as permuting the feature because the model will learn how to use the other features in a better way (or make heavier use of a correlated feature). This is also computationally intensive because you need to retrain. Do NOT try retraining the model

on a sample of the data because your model could have higher error simply because the training set is smaller.

The downsides of this approach is sometimes permutations can yield nonsensical training examples (e.g. shuffling height while keeping weight fixed). Also, if you have two correlated features, they might split up feature importance between themselves.

7 Global Surrogate

We train an interpretable (surrogate) model, like a linear model or decision tree, to approximate a black box model. Basically, we take a dataset of examples, label them with the black box model, and train the surrogate model on the labeled examples. R^2 can tell us how well the surrogate model matches the black box model. If the R^2 is near 1, you can use the surrogate model in place of the black box model. If R^2 is near 0, the surrogate model is not useful.

If you only want a local surrogate, you can sample points such that points near the part of space you care about are sampled with higher weight.

8 Local Surrogate (LIME)

Suppose you are given an example and you want to know why your model made a particular prediction for it. We first generate new examples by randomly perturbing the given example. We then weight each example by its proximity to the given example and train an interpretable model on the (generated example, model prediction for generated example) pairs. Lasso regression is a good surrogate model because we can choose the regularization term so that it does not use too many features in the model.

How do you perturb data examples? For tabular data, we select each feature by drawing from a Gaussian parameterized by the feature mean and feature variance. We measure distance between points using the exponential smoothing kernel (picking the kernel width is hard, especially for high dimensional data). For text data, we randomly remove words. For image data, we find superpixels in the image (clusters of pixels with similar colors) and randomly turn the superpixels on or off.

The biggest problem with LIME is measuring the distance between two data points. In high dimensional data, it's very hard to find a good proximity measure (the exponential kernel does not work well). Also, LIME explanations are not very stable in practice (nearby points sometimes have very different explanations).

9 Shapley Values

Let's imagine the prediction is a payout that the features (game players) compete for. This game theoretical idea is the basis for Shapley values.

Suppose we have an example and a feature of interest. Take a subset of the (feature name, feature value) pairs from the example (except the feature of interest) and call this a coalition. Randomly pick other examples that contain this coalition. Compute the model prediction for each of them, compute the difference from the prediction for the original example, and average the differences. This is the marginal contribution of the feature of interest to the coalition. We can repeat this for all possible coalitions. Thus, the Shapley value is the average of all the marginal contributions to all possible coalitions. Obviously, this is expensive to do because there could be many coalitions. So, we need a way to approximate it.

So, we use Monte-Carlo sampling. Suppose we have an example of interest (call it x) and a feature of interest (call it feature j). We pick a random example (call it z). We pick a random permutation of feature values and apply it to both the random example and example of interest to get z_o and x_o, respectively. Then, we construct two new examples. The first is $x_{+j} = x_{o,1}, ..., x_{o,j-1}, x_{o,j}, z_{o,j+1}, ..., z_{o,p}$ and the

second is $x_{-j} = x_{o,1}, ..., x_{o,j-1}, z_{o,j}, z_{o,j+1}, ..., z_{o,p}$. The marginal contribution is then $\hat{f}(x_{+j}) - \hat{f}(x_{-j})$. We do this M times and average the marginal contributions to get the Shapley value for feature j. We then compute the Shapley value for all features.

Shapley values offer more truthful explanations than LIME, so use them if you need very accurate explanations. It also has solid game theory behind it.

Chapter 6 - Example-Based Explanations

1 Introduction

If examples are interpretable (e.g. images, text), we can select a set of examples and use that an explanation.

2 Counterfactual Explanations

This is an explanation of the form "If X had not happened, Y would not have happened". Basically, we want to know how much the features would have to change in order for the prediction to change (e.g. if you earned two hundred extra dollars a month, you would have received the loan).

How do we generate a counterfactual (x') given the original example (x) and desired target y'? Trial and error changes of x is one option, but is inefficient. A better option is to optimize a loss function that balances the similarity to the target and the closeness of the counterfactual to the original point. We have $L(x, x', y', \lambda) = \lambda \cdot (\hat{f}(x') - y')^2 + d(x, x')$. Alternatively, we can require $|\hat{f}(x') - y'| \leq \epsilon$ for a given ϵ and then optimize $\text{argmin}_{x'} \text{max}_{\lambda} L(x, x', y', \lambda)$. We measure the distance with Manhattan distance weighted by median absolute deviation. That is, $d(x, x') = \sum_{j=1}^{p} \frac{|x_j - x'_j|}{MAD_j}$ where $MAD_j = \text{median}_{i \in \{1...n\}}(|x_{i,j} - \text{median}_{l \in \{1...n\}}(x_{l,j})|)$. This is better than Euclidean distance because it induces sparsity and is resistant to outliers.

To find the counterfactual for a given (x, y', ϵ), we do the following. We sample a random x' and optimize L. Then, while $|\hat{f}(x') - y'| > \epsilon$, we increase λ, and optimize L. Finally, we return the counterfactual x'. We can run this process multiple times to find many counterfactuals.

The counterfactual method does not require access to the model internals or training data, works with non-machine-learning models, and is easy to implement (if you use an optimization library). One disadvantage is that you need to use a good optimizer that can handle categorical variables.

3 Adversarial Examples

An adversarial example is a training example with small perturbations (that a human cannot notice) that make the model make a false prediction. For example, an attacker might generate an adversarial example to trick a computer vision model.

Most research on adversarial examples have been done on deep neural networks for object recognition. These methods take advantage of the gradient, but it is possible to generate adversarial examples in a model agnostic way (you just need to be able to run the model and generate predictions). Basically, the adversarial examples are found by minimizing $loss(\hat{f}(x + r), l) + c \cdot |r|$ where r is the perturbation, l is the desired outcome, x is the original example, and c is a parameter to balance preference between the two terms in the expression. Box-constrained L-BFGS works well to optimize it.

Another way to find adversarial examples is to add some random noise in the direction of the gradient, That is $x' \leftarrow x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$. This attack works well on neural networks that learn a linear relationship between pixel and score.

The one-pixel attack uses differential evolution to generate an adversarial example by changing a single pixel. Differential evolution is a genetic algorithm where we generate random candidate solutions and generate new solutions with random mutations and mixing the best candidates.

You can also print a patch (or 3D print an object) and place it next to an object to trick a deep neural net. The expectation over transformation (EOT) method allows the adversarial attack to work even when the image is transformed (e.g. rotated)

Suppose you expose a model via a web API. Most attacks will not work because they need a gradient. However, there is a way to launch a black box attack. First, you should generate some legitimate examples and get scores for them. Then you should train a surrogate model on the labeled dataset. Then, you should generate synthetic images (via a hueristic that moves pixels to maximize model score variance). Repeat the previous steps many times. Now generate adversarial examples for the surrogate model and use them to attack the web API model.

Some ways to protect against adversarial attacks include retraining your model periodically, using an ensemble of models, and including adversarial training examples in your dataset. However, you should keep adapting and coming up with new ideas because your adversaries are developing new attacks.

4 Prototypes and Criticisms

"A prototype is a data instance that is representative of all the data. A criticism is a data instance that is not well represented by the set of prototypes". The k medoids algorithm can find prototypes.

The maximum mean discrepency (MMD) critic can find both prototypes and criticisms. We greedily find prototypes (i.e. points whose distribution is similar to data distribution) and then greedily find criticisms (i.e. points whose distribution is different than data distribution). We need a kernel function k to measure density (similarity), the MMD metric to measure how similar two distributions are, and the witness function (based on kernel) to tell how different two distributions are at a given point. The MMD is measured as follows (m is number of prototypes, n is number of data points in dataset).

$$MMD^2 = \frac{1}{m^2}\sum_{i,j=1}^{m} k(z_i, z_j) - \frac{2}{mn}\sum_{i,j=1}^{m,n} k(z_i, x_j) + \frac{1}{n^2}\sum_{i,j=1}^{n} k(x_i, x_j) \tag{1}$$

Good prototypes have lower MMD^2 values. A good kernel is the radial basis function $k(x, x') = \exp(\gamma||x - x'||^2)$, where you need to pick γ. We find prototypes by greedily picking prototypes that decrease the MMD^2 value the most. The witness function is

$$witness(x) = \frac{1}{n}\sum_{i=1}^{n} k(x, x_i) - \frac{1}{m}\sum_{j=1}^{m} k(x, z_j) \tag{2}$$

A high absolute value of the witness function indicates a good criticism. We find criticisms by greedily picking points that maximize the absolute witness value plus a regularization term that enforces diversity of criticisms.

The prototypes and criticisms on their own can help you understand your model, but you can also create a simple "nearest prototype" model, which is pretty interpretable. You should run your model on the prototypes and criticisms and manually ensure that the data looks good.

MMD critic is flexible, interpretable, and easy to implement. The problem with the approach is you need to pick the number of prototypes, number of criticisms, and kernel parameter.

5 Influential Instances

An influential training instance is a training example that changes the model a lot when it is deleted and the model is retrained.

An outlier is a point that is far away from the rest of the dataset. It might be a good criticism in the dataset. It might also be an influential instance.

One way to find influential instances is with the deletion approach. One metric is $DFBETA_i = \beta - \beta^{(-i)}$, which is simply the change in the weight vector after deleting instance i. Another metric is Cook's distance, which is $D_i = \frac{\sum_{j=1}^{n}(\hat{y}_j - \hat{y}_j^{(-i)})^2}{p \cdot MSE}$. p is the number of features and MSE is the original mean squared error. Cook's distance does not require a weight vector, unlike DFBETA, but it assumes that the mean squared error is meaningful, which is not true for classification. Another option is to use $\text{Influence}^{(-i)} = \frac{1}{n}\sum_{j=1}^{n}|\hat{y}_j - \hat{y}_j^{(-i)}|$. We can also see how removing example i affects example j with $\text{Influence}_j^{(-i)} = |\hat{y}_j - \hat{y}_j^{(-i)}|$. To use these metrics, for each point, remove it, retrain model, and compute the influence of the point. The most influential points are worth investigating. We could also train an interpretable model to predict the influence of a data point to see what exactly makes influential data points influential. We could also pick a training example and see which other point's removal influences its prediction the most.

Of course, the key problem with the deletion approach is that you have to retrain the model once for each training example. To overcome this problem, we can use influence functions, but we need to have a twice differentiable loss function in order to do this.

Suppose we upweight point z. We get new parameters

$$\hat{\theta}_{\epsilon,z} = \text{argmin}_{\theta \in \Theta}(1-\epsilon)\frac{1}{n}\sum_{i=1}^{n} L(z_i, \theta) + \epsilon L(z, \theta) \tag{3}$$

The influence of upweighting z is then

$$I_{up,params}(z) = \frac{d\hat{\theta}_{\epsilon,z}}{d\epsilon}]_{\epsilon=0} = -H_{\hat{\theta}}^{-1}\nabla_\theta L(z, \hat{\theta}) \tag{4}$$

We can estimate the Hessian matrix with $H_{\hat{\theta}} \approx \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta^2 L(z_i, \theta)$ (there are tricks to efficiently compute this). We can then approximate $\hat{\theta}_{-z} \approx \hat{\theta} - \frac{1}{n}I_{up,params}(z)$ and use these new parameters. If we want to see how upweighting z impacts the prediction of z_{test}, we can do it directly instead of first computing $\hat{\theta}_{-z}$ as follows $I_{up,loss}(z, z_{test}) = -\nabla_\theta L(z_{test}, \hat{\theta})^T H_{\hat{\theta}}^{-1}\nabla_\theta L(z, \hat{\theta})$.

The deletion approach and influence functions allow us to compute the influence of a given point and the influence of a given point on another given point.

We can use this to figure out what training examples influence the model most overall, what makes influential instances influential (train a surrogate interpretable model), and debug why we made a wrong prediction for a test point (see the most influential training point that changes its prediction). We can also spot check influential instances in case they are actually bad data.

Influential instances are great for debugging, model agnostic, and can even be used to generate adversarial examples. Instead of looking at change in loss, change in prediction, you can look at the most influential points that affect feature importance or other metrics.

The downsides are that the deletion method is computationally expensive and influence functions only work on twice differentiable loss functions (so you cannot use them for tree based methods). If you don't have a differentiable loss, try approximating it with a differentiable one. Another limitation is that we can only consider deleting points one at a time, rather than in groups.

Chapter 7 - A Look into the Crystal Ball

1 Introduction

Let us assume that all informatin will be be digitized, automatable tasks will be automated, and that for interesting problems it is not possible to exactly specify our goal with all possible constraints (a corporation tasked only with maximizing profits might do crazy things like pollute rivers, hire child laborers, and fund militias in developing countries to achieve this).

With this in hand, let's predict the future.

2 The Future of Machine Learning

Machine learning adoption will advance, slowly but surely. It takes time for large organizations to figure out good team structures, good data architectures, and good systems to enable data scientists.

Machine learning will become easier to use.

More and more tasks will be formulated as prediction problems so that machine learning can be used.

As machine learning is used in high stakes tasks and regulations are created, interpretability will become more important. Many people today do not use machine learning simply because it is not interpretable.

3 The Future of Interpretability

Model-agnostic interpretability tools will dominate because they are the most portable. But, intrinsically interpretable methods might have a use for some cases.

Most systems will include automated interpretability in the same way they include automatic hyperparameter selection and automatic ensembling.

People will be able to train machine learning models even without knowing how to program.

People will use interpretable machine learning to glean insights about their data, not just for using the model.

Traditional methods, like linear models, make too many unrealistic assumptions, so black box models will become more widely used because they will achieve better performance.

Traditional statistical techniques like hypothesis tests and confidence intervals will be adapted for black box methods.

Data scientists will automate away many of their tasks.

Robots and programs will have user interfaces to explain why they make certain decisions.

Interpretability might help us understand more about intelligence in general.