Interpretable Models

The easiest way to achieve interpretability is to use only a subset of algorithms that create interpretable models. Linear regression, logistic regression and the decision tree are commonly used interpretable models.

In the following chapters we will talk about these models. Not in detail, only the basics, because there is already a ton of books, videos, tutorials, papers and more material available. We will focus on how to interpret the models. The book discusses linear regression, logistic regression, other linear regression extensions, decision trees, decision rules and the RuleFit algorithm in more detail. It also lists other interpretable models.

All interpretable models explained in this book are interpretable on a modular level, with the exception of the k-nearest neighbors method. The following table gives an overview of the interpretable model types and their properties. A model is linear if the association between features and target is modelled linearly. A model with monotonicity constraints ensures that the relationship between a feature and the target outcome always goes in the same direction over the entire range of the feature: An increase in the feature value either always leads to an increase or always to a decrease in the target outcome. Monotonicity is useful for the interpretation of a model because it makes it easier to understand a relationship. Some models can automatically include interactions between features to predict the target outcome. You can include interactions in any type of model by manually creating interaction features. Interactions can improve predictive performance, but too many or too complex interactions can hurt interpretability. Some models handle only regression, some only classification, and still others both.

From this table, you can select a suitable interpretable model for your task, either regression (regr) or classification (class):

Algorithm	Linear	Monotone	Interaction	Task
Linear regression	Yes	Yes	No	regr
Logistic regression	No	Yes	No	class
Decision trees	No	Some	Yes	class,regr
RuleFit	Yes	No	Yes	class,regr
Naive Bayes	No	Yes	No	class
k-nearest neighbors	No	No	No	class,regr

Linear Regression

A linear regression model predicts the target as a weighted sum of the feature inputs. The linearity of the learned relationship makes the interpretation easy. Linear regression models have long been used by statisticians, computer scientists and other people who tackle quantitative problems.

Linear models can be used to model the dependence of a regression target y on some features x. The learned relationships are linear and can be written for a single instance i as follows:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \epsilon$$

The predicted outcome of an instance is a weighted sum of its p features. The betas (β_j) represent the learned feature weights or coefficients. The first weight in the sum (β_0) is called the intercept and is not multiplied with a feature. The epsilon (ϵ) is the error we still make, i.e. the difference between the prediction and the actual outcome. These errors are assumed to follow a Gaussian distribution, which means that we make errors in both negative and positive directions and make many small errors and few large errors.

Various methods can be used to estimate the optimal weight. The ordinary least squares method is usually used to find the weights that minimize the squared differences between the actual and the estimated outcomes:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} \left(y^{(i)} - \left(\beta_0 + \sum_{j=1}^{p} \beta_j x_j^{(i)} \right) \right)^2$$

We will not discuss in detail how the optimal weights can be found, but if you are interested, you can read chapter 3.2 of the book "The Elements of Statistical Learning" (Friedman, Hastie and Tibshirani 2009)[35] or one of the other online resources on linear regression models.

The biggest advantage of linear regression models is linearity: It makes the estimation procedure simple and, most importantly, these linear equations have an easy to understand interpretation on a modular level (i.e. the weights). This is one of the main reasons why the linear model and all similar models are so widespread in academic fields such as medicine, sociology, psychology, and many other quantitative research fields. For example, in the medical field, it is not only important to predict the clinical outcome of a patient, but also to quantify the influence of the drug and at the same time take sex, age, and other features into account in an interpretable way.

Estimated weights come with confidence intervals. A confidence interval is a range for the weight estimate that covers the "true" weight with a certain confidence. For example, a 95% confidence interval for a weight of 2 could range from 1 to 3. The interpretation of this interval would be: If we repeated the estimation 100 times with newly sampled data, the confidence interval would include

[35] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. "The elements of statistical learning". www.web.stanford.edu/~hastie/ElemStatLearn/ (2009).

the true weight in 95 out of 100 cases, given that the linear regression model is the correct model for the data.

Whether the model is the "correct" model depends on whether the relationships in the data meet certain assumptions, which are linearity, normality, homoscedasticity, independence, fixed features, and absence of multicollinearity.

Linearity
The linear regression model forces the prediction to be a linear combination of features, which is both its greatest strength and its greatest limitation. Linearity leads to interpretable models. Linear effects are easy to quantify and describe. They are additive, so it is easy to separate the effects. If you suspect feature interactions or a nonlinear association of a feature with the target value, you can add interaction terms or use regression splines.

Normality
It is assumed that the target outcome given the features follows a normal distribution. If this assumption is violated, the estimated confidence intervals of the feature weights are invalid.

Homoscedasticity (constant variance)
The variance of the error terms is assumed to be constant over the entire feature space. Suppose you want to predict the value of a house given the living area in square meters. You estimate a linear model that assumes that, regardless of the size of the house, the error around the predicted response has the same variance. This assumption is often violated in reality. In the house example, it is plausible that the variance of error terms around the predicted price is higher for larger houses, since prices are higher and there is more room for price fluctuations. Suppose the average error (difference between predicted and actual price) in your linear regression model is 50,000 Euros. If you assume homoscedasticity, you assume that the average error of 50,000 is the same for houses that cost 1 million and for houses that cost only 40,000. This is unreasonable because it would mean that we can expect negative house prices.

Independence
It is assumed that each instance is independent of any other instance. If you perform repeated measurements, such as multiple blood tests per patient, the data points are not independent. For dependent data you need special linear regression models, such as mixed effect models or GEEs. If you use the "normal" linear regression model, you might draw wrong conclusions from the model.

Fixed features
The input features are considered "fixed". Fixed means that they are treated as "given constants" and not as statistical variables. This implies that they are free of measurement errors. This is a rather unrealistic assumption. Without that assumption, however, you would have to fit very complex measurement error models that account for the measurement errors of your input features. And usually you do not want to do that.

Absence of multicollinearity
You do not want strongly correlated features, because this messes up the estimation of the weights. In a situation where two features are strongly correlated, it becomes problematic to estimate the

weights because the feature effects are additive and it becomes indeterminable to which of the correlated features to attribute the effects.

Interpretation

The interpretation of a weight in the linear regression model depends on the type of the corresponding feature.

- Numerical feature: Increasing the numerical feature by one unit changes the estimated outcome by its weight. An example of a numerical feature is the size of a house.
- Binary feature: A feature that takes one of two possible values for each instance. An example is the feature "House comes with a garden". One of the values counts as the reference category (in some programming languages encoded with 0), such as "No garden". Changing the feature from the reference category to the other category changes the estimated outcome by the feature's weight.
- Categorical feature with multiple categories: A feature with a fixed number of possible values. An example is the feature "floor type", with possible categories "carpet", "laminate" and "parquet". A solution to deal with many categories is the one-hot-encoding, meaning that each category has its own binary column. For a categorical feature with L categories, you only need L-1 columns, because the L-th column would have redundant information (e.g. when columns 1 to L-1 all have value 0 for one instance, we know that the categorical feature of this instance takes on category L). The interpretation for each category is then the same as the interpretation for binary features. Some languages, such as R, allow you to encode categorical features in various ways, as described later in this chapter.
- Intercept β_0: The intercept is the feature weight for the "constant feature", which is always 1 for all instances. Most software packages automatically add this "1"-feature to estimate the intercept. The interpretation is: For an instance with all numerical feature values at zero and the categorical feature values at the reference categories, the model prediction is the intercept weight. The interpretation of the intercept is usually not relevant because instances with all features values at zero often make no sense. The interpretation is only meaningful when the features have been standardised (mean of zero, standard deviation of one). Then the intercept reflects the predicted outcome of an instance where all features are at their mean value.

The interpretation of the features in the linear regression model can be automated by using following text templates.

Interpretation of a Numerical Feature

An increase of feature x_k by one unit increases the prediction for y by β_k units when all other feature values remain fixed.

Interpretation of a Categorical Feature

Changing feature x_k from the reference category to the other category increases the prediction for y by β_k when all other features remain fixed.

Another important measurement for interpreting linear models is the R-squared measurement. R-squared tells you how much of the total variance of your target outcome is explained by the model. The higher R-squared, the better your model explains the data. The formula for calculating R-squared is:

$$R^2 = 1 - SSE/SST$$

SSE is the squared sum of the error terms:

$$SSE = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

SST is the squared sum of the data variance:

$$SST = \sum_{i=1}^{n}(y^{(i)} - \bar{y})^2$$

The SSE tells you how much variance remains after fitting the linear model, which is measured by the squared differences between the predicted and actual target values. SST is the total variance of the target outcome. R-squared tells you how much of your variance can be explained by the linear model. R-squared ranges between 0 for models where the model does not explain the data at all and 1 for models that explain all of the variance in your data.

There is a catch, because R-squared increases with the number of features in the model, even if they do not contain any information about the target value at all. Therefore, it is better to use the adjusted R-squared, which accounts for the number of features used in the model. Its calculation is:

$$\bar{R}^2 = R^2 - (1 - R^2)\frac{p}{n - p - 1}$$

where p is the number of features and n the number of instances.

It is not meaningful to interpret a model with very low (adjusted) R-squared, because such a model basically does not explain much of the variance. Any interpretation of the weights would not be meaningful.

Feature Importance

The importance of a feature in a linear regression model can be measured by the absolute value of its t-statistic. The t-statistic is the estimated weight scaled with its standard error.

$$t_{\hat{\beta}_j} = \frac{\hat{\beta}_j}{SE(\hat{\beta}_j)}$$

Let us examine what this formula tells us: The importance of a feature increases with increasing

weight. This makes sense. The more variance the estimated weight has (= the less certain we are about the correct value), the less important the feature is. This also makes sense.

Example

In this example, we use the linear regression model to predict the number of rented bikes on a particular day, given weather and calendar information. For the interpretation, we examine the estimated regression weights. The features consist of numerical and categorical features. For each feature, the table shows the estimated weight, the standard error of the estimate (SE), and the absolute value of the t-statistic (|t|).

	Weight	SE	t
(Intercept)	2399.4	238.3	10.1
seasonSUMMER	899.3	122.3	7.4
seasonFALL	138.2	161.7	0.9
seasonWINTER	425.6	110.8	3.8
holidayHOLIDAY	-686.1	203.3	3.4
workingdayWORKING DAY	124.9	73.3	1.7
weathersitMISTY	-379.4	87.6	4.3
weathersitRAIN/SNOW/STORM	-1901.5	223.6	8.5
temp	110.7	7.0	15.7
hum	-17.4	3.2	5.5
windspeed	-42.5	6.9	6.2
days_since_2011	4.9	0.2	28.5

Interpretation of a numerical feature (temperature): An increase of the temperature by 1 degree Celsius increases the predicted number of bicycles by 110.7, when all other features remain fixed.

Interpretation of a categorical feature ("weathersit"): The estimated number of bicycles is -1901.5 lower when it is raining, snowing or stormy, compared to good weather – again assuming that all other features do not change. When the weather is misty, the predicted number of bicycles is -379.4 lower compared to good weather, given all other features remain the same.

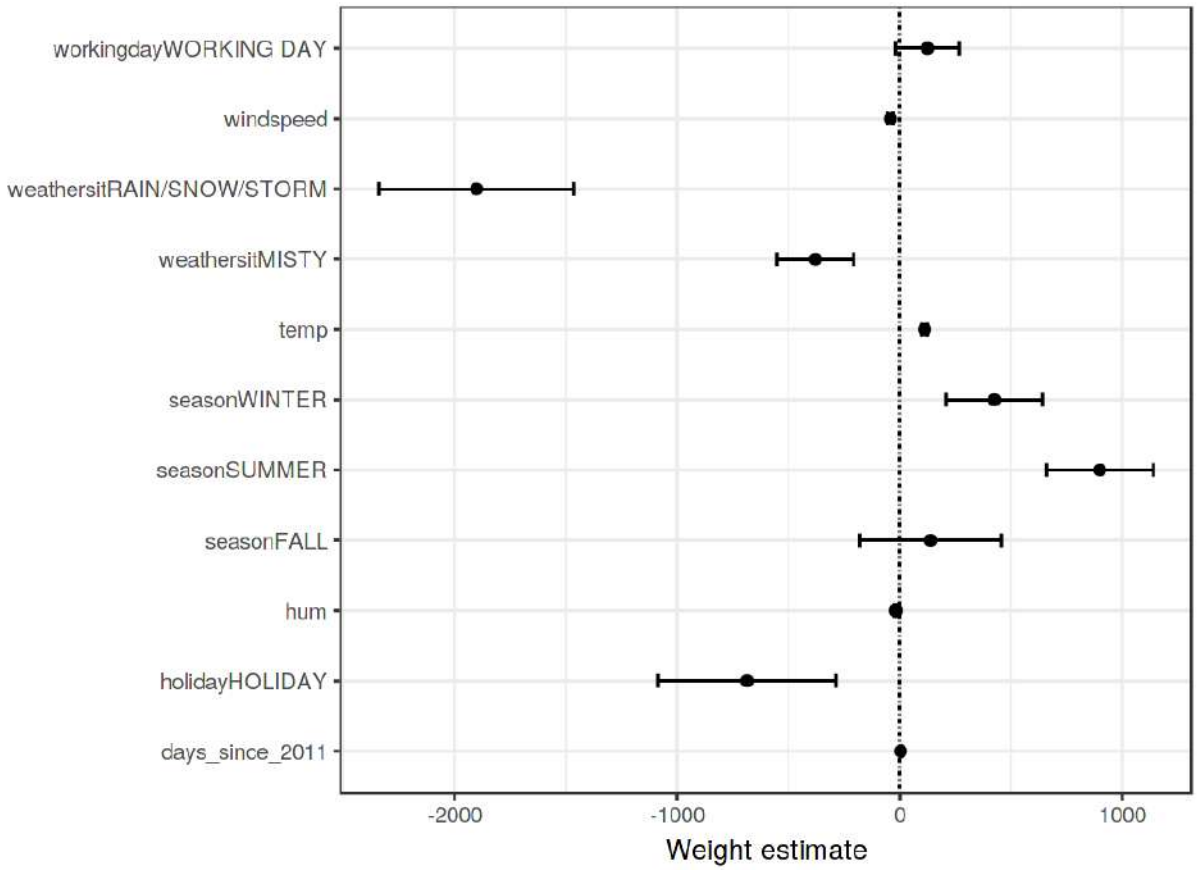
All the interpretations always come with the footnote that "all other features remain the same". This is because of the nature of linear regression models. The predicted target is a linear combination of the weighted features. The estimated linear equation is a hyperplane in the feature/target space (a simple line in the case of a single feature). The weights specify the slope (gradient) of the hyperplane in each direction. The good side is that the additivity isolates the interpretation of an individual feature effect from all other features. That is possible because all the feature effects (= weight times feature value) in the equation are combined with a plus. On the bad side of things, the interpretation ignores the joint distribution of the features. Increasing one feature, but not changing another, can lead to unrealistic or at least unlikely data points. For example increasing the number of rooms might be unrealistic without also increasing the size of a house.

Visual Interpretation

Various visualizations make the linear regression model easy and quick to grasp for humans.

Weight Plot

The information of the weight table (weight and variance estimates) can be visualized in a weight plot. The following plot shows the results from the previous linear regression model.



Weights are displayed as points and the 95% confidence intervals as lines.

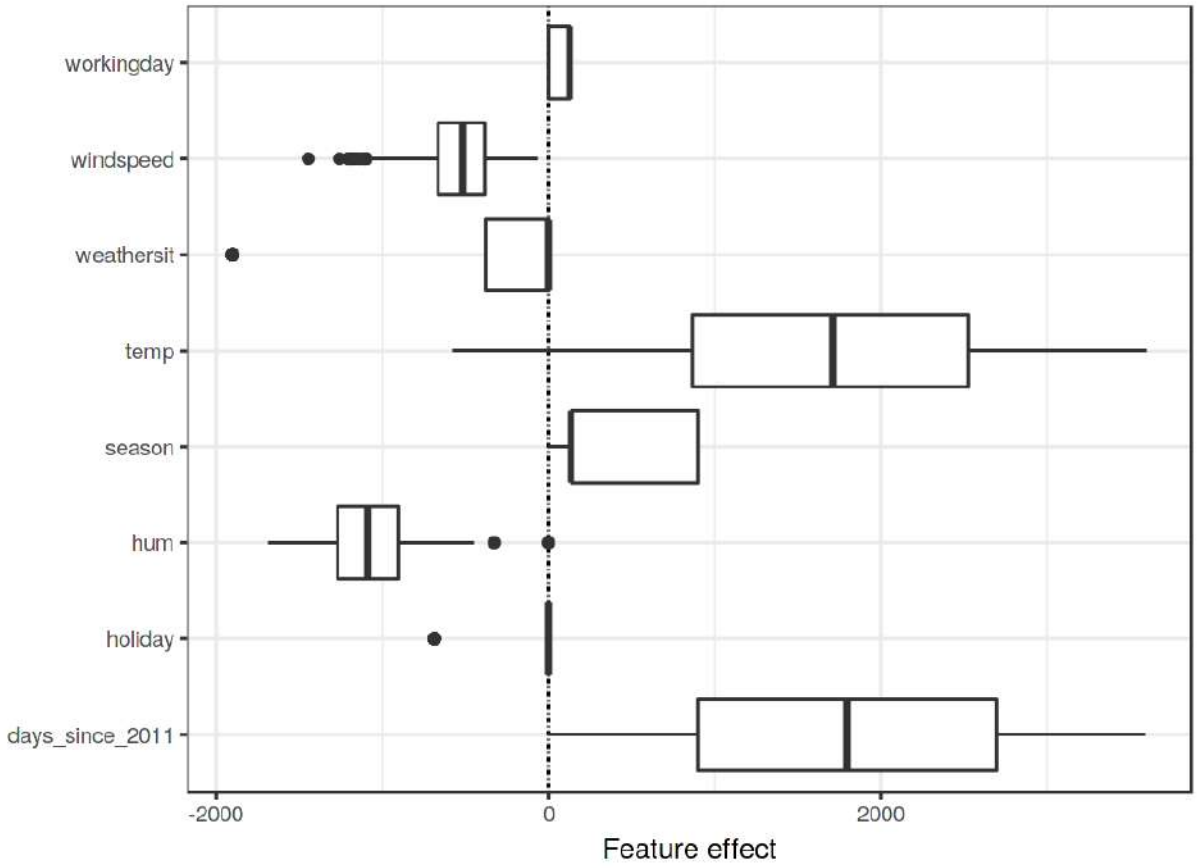
The weight plot shows that rainy/snowy/stormy weather has a strong negative effect on the predicted number of bikes. The weight of the working day feature is close to zero and zero is included in the 95% interval, which means that the effect is not statistically significant. Some confidence intervals are very short and the estimates are close to zero, yet the feature effects were statistically significant. Temperature is one such candidate. The problem with the weight plot is that the features are measured on different scales. While for the weather the estimated weight reflects the difference between good and rainy/stormy/snowy weather, for temperature it only reflects an increase of 1 degree Celsius. You can make the estimated weights more comparable by scaling the features (zero mean and standard deviation of one) before fitting the linear model.

Effect Plot

The weights of the linear regression model can be more meaningfully analyzed when they are multiplied by the actual feature values. The weights depend on the scale of the features and will be different if you have a feature that measures e.g. a person's height and you switch from meter to centimeter. The weight will change, but the actual effects in your data will not. It is also important to know the distribution of your feature in the data, because if you have a very low variance, it means that almost all instances have similar contribution from this feature. The effect plot can help you understand how much the combination of weight and feature contributes to the predictions in your data. Start by calculating the effects, which is the weight per feature times the feature value of an instance:

$$\text{effect}_j^{(i)} = w_j x_j^{(i)}$$

The effects can be visualized with boxplots. A box in a boxplot contains the effect range for half of your data (25% to 75% effect quantiles). The vertical line in the box is the median effect, i.e. 50% of the instances have a lower and the other half a higher effect on the prediction. The horizontal lines extend to $\pm 1.58 \text{IQR}/\sqrt{n}$, with IQR being the inter quartile range (75% quantile minus 25% quantile). The dots are outliers. The categorical feature effects can be summarized in a single boxplot, compared to the weight plot, where each category has its own row.



The feature effect plot shows the distribution of effects (= feature value times feature weight) across the data per feature.

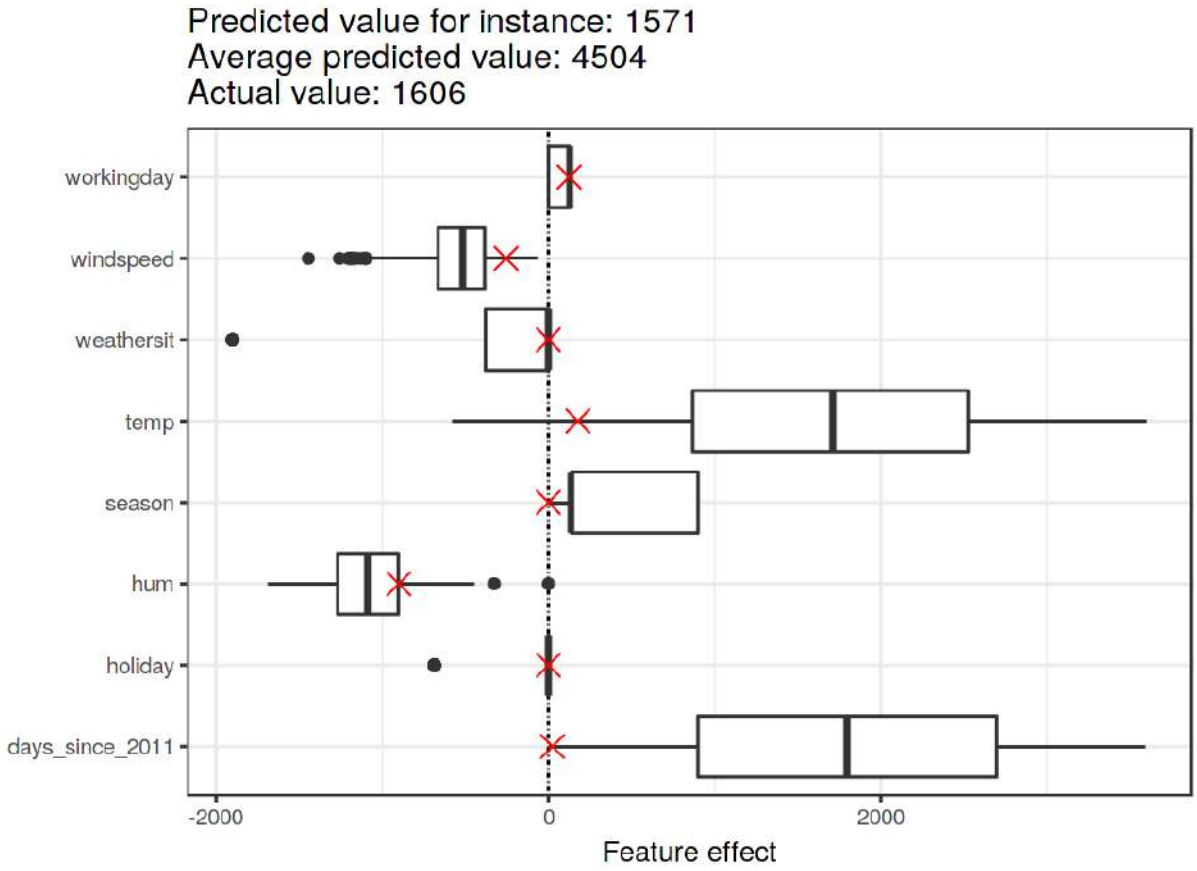
The largest contributions to the expected number of rented bicycles comes from the temperature feature and the days feature, which captures the trend of bike rentals over time. The temperature has a broad range of how much it contributes to the prediction. The day trend feature goes from zero to large positive contributions, because the first day in the dataset (01.01.2011) has a very small trend effect and the estimated weight for this feature is positive (4.93). This means that the effect increases with each day and is highest for the last day in the dataset (31.12.2012). Note that for effects with a negative weight, the instances with a positive effect are those that have a negative feature value. For example, days with a high negative effect of windspeed are the ones with high wind speeds.

Explain Individual Predictions

How much has each feature of an instance contributed to the prediction? This can be answered by computing the effects for this instance. An interpretation of instance-specific effects only makes sense in comparison to the distribution of the effect for each feature. We want to explain the prediction of the linear model for the 6-th instance from the bicycle dataset. The instance has the following feature values.

Feature	Value
season	SPRING
yr	2011
mnth	JAN
holiday	NO HOLIDAY
weekday	THU
workingday	WORKING DAY
weathersit	GOOD
temp	1.604356
hum	51.8261
windspeed	6.000868
cnt	1606
days_since_2011	5

To obtain the feature effects of this instance, we have to multiply its feature values by the corresponding weights from the linear regression model. For the value "WORKING DAY" of feature "workingday", the effect is, 124.9. For a temperature of 1.6 degrees Celsius, the effect is 177.6. We add these individual effects as crosses to the effect plot, which shows us the distribution of the effects in the data. This allows us to compare the individual effects with the distribution of effects in the data.



The effect plot for one instance shows the effect distribution and highlights the effects of the instance of interest.

If we average the predictions for the training data instances, we get an average of 4504. In comparison, the prediction of the 6-th instance is small, since only 1571 bicycle rents are predicted. The effect plot reveals the reason why. The boxplots show the distributions of the effects for all instances of the dataset, the red crosses show the effects for the 6-th instance. The 6-th instance has a low temperature effect because on this day the temperature was 2 degrees, which is low compared to most other days (and remember that the weight of the temperature feature is positive). Also, the effect of the trend feature "days_since_2011" is small compared to the other data instances because this instance is from early 2011 (5 days) and the trend feature also has a positive weight.

Encoding of Categorical Features

There are several ways to encode a categorical feature, and the choice influences the interpretation of the weights.

The standard in linear regression models is treatment coding, which is sufficient in most cases. Using different encodings boils down to creating different (design) matrices from a single column with the categorical feature. This section presents three different encodings, but there are many more. The example used has six instances and a categorical feature with three categories. For the first two

instances, the feature takes category A; for instances three and four, category B; and for the last two instances, category C.

Treatment coding

In treatment coding, the weight per category is the estimated difference in the prediction between the corresponding category and the reference category. The intercept of the linear model is the mean of the reference category (when all other features remain the same). The first column of the design matrix is the intercept, which is always 1. Column two indicates whether instance i is in category B, column three indicates whether it is in category C. There is no need for a column for category A, because then the linear equation would be overspecified and no unique solution for the weights can be found. It is sufficient to know that an instance is neither in category B or C.

Feature matrix: $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$

Effect coding

The weight per category is the estimated y-difference from the corresponding category to the overall mean (given all other features are zero or the reference category). The first column is used to estimate the intercept. The weight β_0 associated with the intercept represents the overall mean and β_1, the weight for column two, is the difference between the overall mean and category B. The total effect of category B is $\beta_0 + \beta_1$. The interpretation for category C is equivalent. For the reference category A, $-(\beta_1 + \beta_2)$ is the difference to the overall mean and $\beta_0 - (\beta_1 + \beta_2)$ the overall effect.

Feature matrix: $\begin{pmatrix} 1 & -1 & -1 \\ 1 & -1 & -1 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$

Dummy coding

The β per category is the estimated mean value of y for each category (given all other feature values are zero or the reference category). Note that the intercept has been omitted here so that a unique solution can be found for the linear model weights.

Feature matrix: $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

If you want to dive a little deeper into the different encodings of categorical features, checkout this overview webpage[36] and this blog post[37].

Do Linear Models Create Good Explanations?

Judging by the attributes that constitute a good explanation, as presented in the Human-Friendly Explanations chapter, linear models do not create the best explanations. They are contrastive, but the reference instance is a data point where all numerical features are zero and the categorical features are at their reference categories. This is usually an artificial, meaningless instance that is unlikely to occur in your data or reality. There is an exception: If all numerical features are mean centered (feature minus mean of feature) and all categorical features are effect coded, the reference instance is the data point where all the features take on the mean feature value. This might also be a non-existent data point, but it might at least be more likely or more meaningful. In this case, the weights times the feature values (feature effects) explain the contribution to the predicted outcome contrastive to the "mean-instance". Another aspect of a good explanation is selectivity, which can be achieved in linear models by using less features or by training sparse linear models. But by default, linear models do not create selective explanations. Linear models create truthful explanations, as long as the linear equation is an appropriate model for the relationship between features and outcome. The more non-linearities and interactions there are, the less accurate the linear model will be and the less truthful the explanations become. Linearity makes the explanations more general and simpler. The linear nature of the model, I believe, is the main factor why people use linear models for explaining relationships.

Sparse Linear Models

The examples of the linear models that I have chosen all look nice and neat, do they not? But in reality you might not have just a handful of features, but hundreds or thousands. And your linear regression models? Interpretability goes downhill. You might even find yourself in a situation where there are more features than instances, and you cannot fit a standard linear model at all. The good news is that there are ways to introduce sparsity (= few features) into linear models.

Lasso

Lasso is an automatic and convenient way to introduce sparsity into the linear regression model. Lasso stands for "least absolute shrinkage and selection operator" and, when applied in a linear regression model, performs feature selection and regularization of the selected feature weights. Let us consider the minimization problem that the weights optimize:

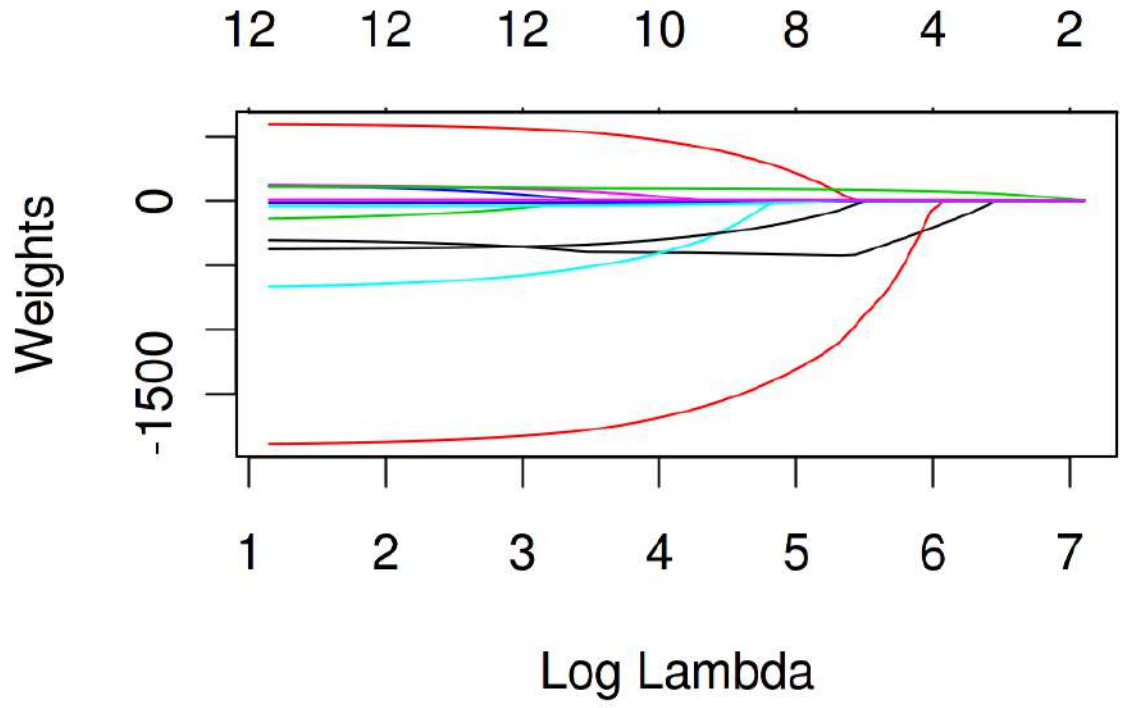
$$min_{\boldsymbol{\beta}} \left(\frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - x_i^T \boldsymbol{\beta})^2 \right)$$

[36] http://stats.idre.ucla.edu/r/library/r-library-contrast-coding-systems-for-categorical-variables/
[37] http://heidiseibold.github.io/page7/

Lasso adds a term to this optimization problem.

$$min_{\boldsymbol{\beta}} \left(\frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - x_i^T \boldsymbol{\beta})^2 + \lambda ||\boldsymbol{\beta}||_1 \right)$$

The term $||\boldsymbol{\beta}||_1$, the L1-norm of the feature vector, leads to a penalization of large weights. Since the L1-norm is used, many of the weights receive an estimate of 0 and the others are shrunk. The parameter lambda (λ) controls the strength of the regularizing effect and is usually tuned by cross-validation. Especially when lambda is large, many weights become 0. The feature weights can be visualized as a function of the penalty term lambda. Each feature weight is represented by a curve in the following figure.



With increasing penalty of the weights, fewer and fewer features receive a non-zero weight estimate. These curves are also called regularization paths. The number above the plot is the number of non-zero weights.

What value should we choose for lambda? If you see the penalization term as a tuning parameter, then you can find the lambda that minimizes the model error with cross-validation. You can also consider lambda as a parameter to control the interpretability of the model. The larger the penalization, the fewer features are present in the model (because their weights are zero) and the better the model can be interpreted.

Example with Lasso

We will predict bicycle rentals using Lasso. We set the number of features we want to have in the model beforehand. Let us first set the number to 2 features:

	Weight
seasonSPRING	0.00
seasonSUMMER	0.00
seasonFALL	0.00
seasonWINTER	0.00
holidayHOLIDAY	0.00
workingdayWORKING DAY	0.00
weathersitMISTY	0.00
weathersitRAIN/SNOW/STORM	0.00
temp	52.33
hum	0.00
windspeed	0.00
days_since_2011	2.15

The first two features with non-zero weights in the Lasso path are temperature ("temp") and the time trend ("days_since_2011").

Now, let us select 5 features:

	Weight
seasonSPRING	-389.99
seasonSUMMER	0.00
seasonFALL	0.00
seasonWINTER	0.00
holidayHOLIDAY	0.00
workingdayWORKING DAY	0.00
weathersitMISTY	0.00
weathersitRAIN/SNOW/STORM	-862.27
temp	85.58
hum	-3.04
windspeed	0.00
days_since_2011	3.82

Note that the weights for "temp" and "days_since_2011" differ from the model with two features. The reason for this is that by decreasing lambda even features that are already "in" the model are penalized less and may get a larger absolute weight. The interpretation of the Lasso weights corresponds to the interpretation of the weights in the linear regression model. You only need to pay attention to whether the features are standardized or not, because this affects the weights. In this example, the features were standardized by the software, but the weights were automatically transformed back for us to match the original feature scales.

Other methods for sparsity in linear models

A wide spectrum of methods can be used to reduce the number of features in a linear model.

Pre-processing methods:

- Manually selected features: You can always use expert knowledge to select or discard some features. The big drawback is that it cannot be automated and you need to have access to someone who understands the data.
- Univariate selection: An example is the correlation coefficient. You only consider features that exceed a certain threshold of correlation between the feature and the target. The disadvantage is that it only considers the features individually. Some features might not show a correlation until the linear model has accounted for some other features. Those ones you will miss with univariate selection methods.

Step-wise methods:

- Forward selection: Fit the linear model with one feature. Do this with each feature. Select the model that works best (e.g. highest R-squared). Now again, for the remaining features, fit different versions of your model by adding each feature to your current best model. Select the one that performs best. Continue until some criterion is reached, such as the maximum number of features in the model.
- Backward selection: Similar to forward selection. But instead of adding features, start with the model that contains all features and try out which feature you have to remove to get the highest performance increase. Repeat this until some stopping criterion is reached.

I recommend using Lasso, because it can be automated, considers all features simultaneously, and can be controlled via lambda. It also works for the logistic regression model for classification.

Advantages

The modeling of the predictions as a **weighted sum** makes it transparent how predictions are produced. And with Lasso we can ensure that the number of features used remains small.

Many people use linear regression models. This means that in many places it is **accepted** for predictive modeling and doing inference. There is a **high level of collective experience and expertise**, including teaching materials on linear regression models and software implementations. Linear regression can be found in R, Python, Java, Julia, Scala, Javascript, ...

Mathematically, it is straightforward to estimate the weights and you have a **guarantee to find optimal weights** (given all assumptions of the linear regression model are met by the data).

Together with the weights you get confidence intervals, tests, and solid statistical theory. There are also many extensions of the linear regression model (see chapter on GLM, GAM and more).

Disadvantages

Linear regression models can only represent linear relationships, i.e. a weighted sum of the input features. Each **nonlinearity or interaction has to be hand-crafted** and explicitly given to the model as an input feature.

Linear models are also often **not that good regarding predictive performance**, because the relationships that can be learned are so restricted and usually oversimplify how complex reality is.

The interpretation of a weight **can be unintuitive** because it depends on all other features. A feature with high positive correlation with the outcome y and another feature might get a negative weight in the linear model, because, given the other correlated feature, it is negatively correlated with y in the high-dimensional space. Completely correlated features make it even impossible to find a unique solution for the linear equation. An example: You have a model to predict the value of a house and have features like number of rooms and size of the house. House size and number of rooms are highly correlated: the bigger a house is, the more rooms it has. If you take both features into a linear model, it might happen, that the size of the house is the better predictor and gets a large positive weight. The number of rooms might end up getting a negative weight, because, given that a house has the same size, increasing the number of rooms could make it less valuable or the linear equation becomes less stable, when the correlation is too strong.

Logistic Regression

Logistic regression models the probabilities for classification problems with two possible outcomes. It's an extension of the linear regression model for classification problems.

What is Wrong with Linear Regression for Classification?

The linear regression model can work well for regression, but fails for classification. Why is that? In case of two classes, you could label one of the classes with 0 and the other with 1 and use linear regression. Technically it works and most linear model programs will spit out weights for you. But there are a few problems with this approach:

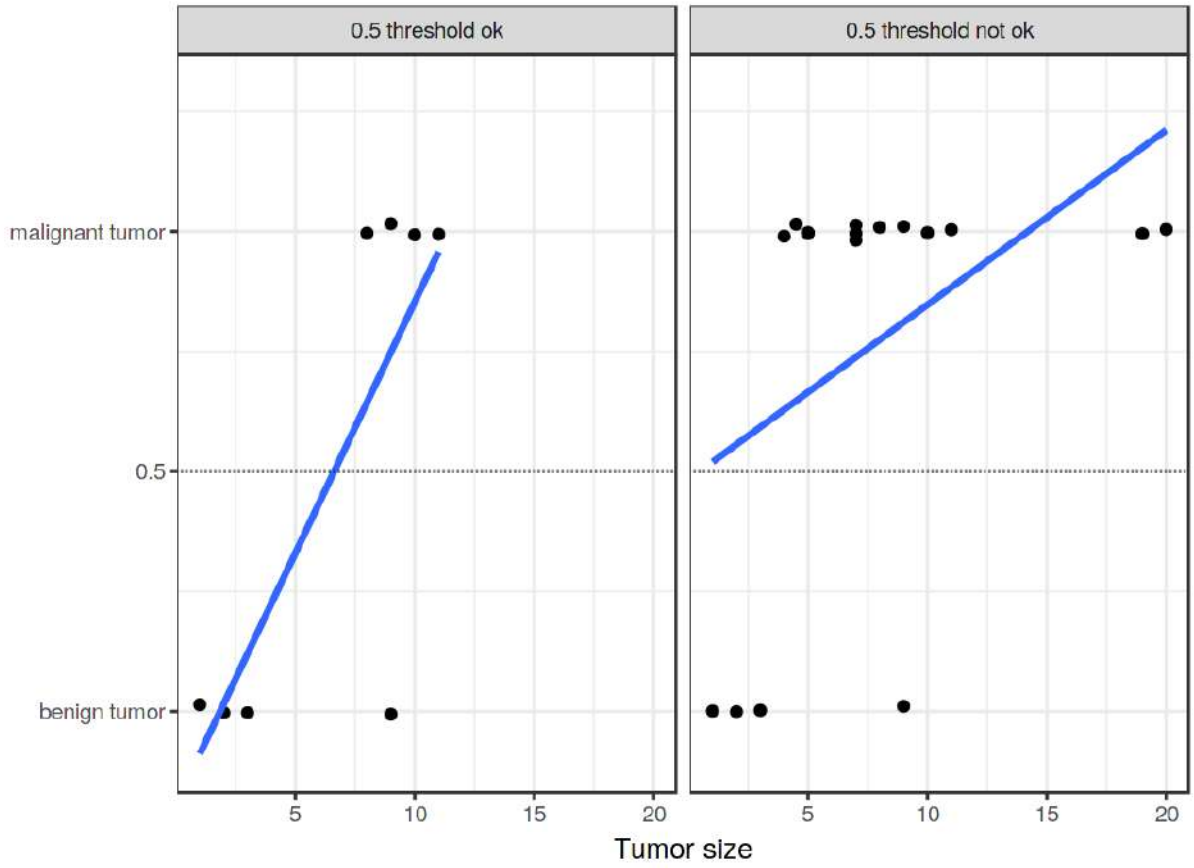
A linear model does not output probabilities, but it treats the classes as numbers (0 and 1) and fits the best hyperplane (for a single feature, it is a line) that minimizes the distances between the points and the hyperplane. So it simply interpolates between the points, and you cannot interpret it as probabilities.

A linear model also extrapolates and gives you values below zero and above one. This is a good sign that there might be a smarter approach to classification.

Since the predicted outcome is not a probability, but a linear interpolation between points, there is no meaningful threshold at which you can distinguish one class from the other. A good illustration of this issue has been given on Stackoverflow[38].

Linear models do not extend to classification problems with multiple classes. You would have to start labeling the next class with 2, then 3, and so on. The classes might not have any meaningful order, but the linear model would force a weird structure on the relationship between the features and your class predictions. The higher the value of a feature with a positive weight, the more it contributes to the prediction of a class with a higher number, even if classes that happen to get a similar number are not closer than other classes.

[38] https://stats.stackexchange.com/questions/22381/why-not-approach-classification-through-regression



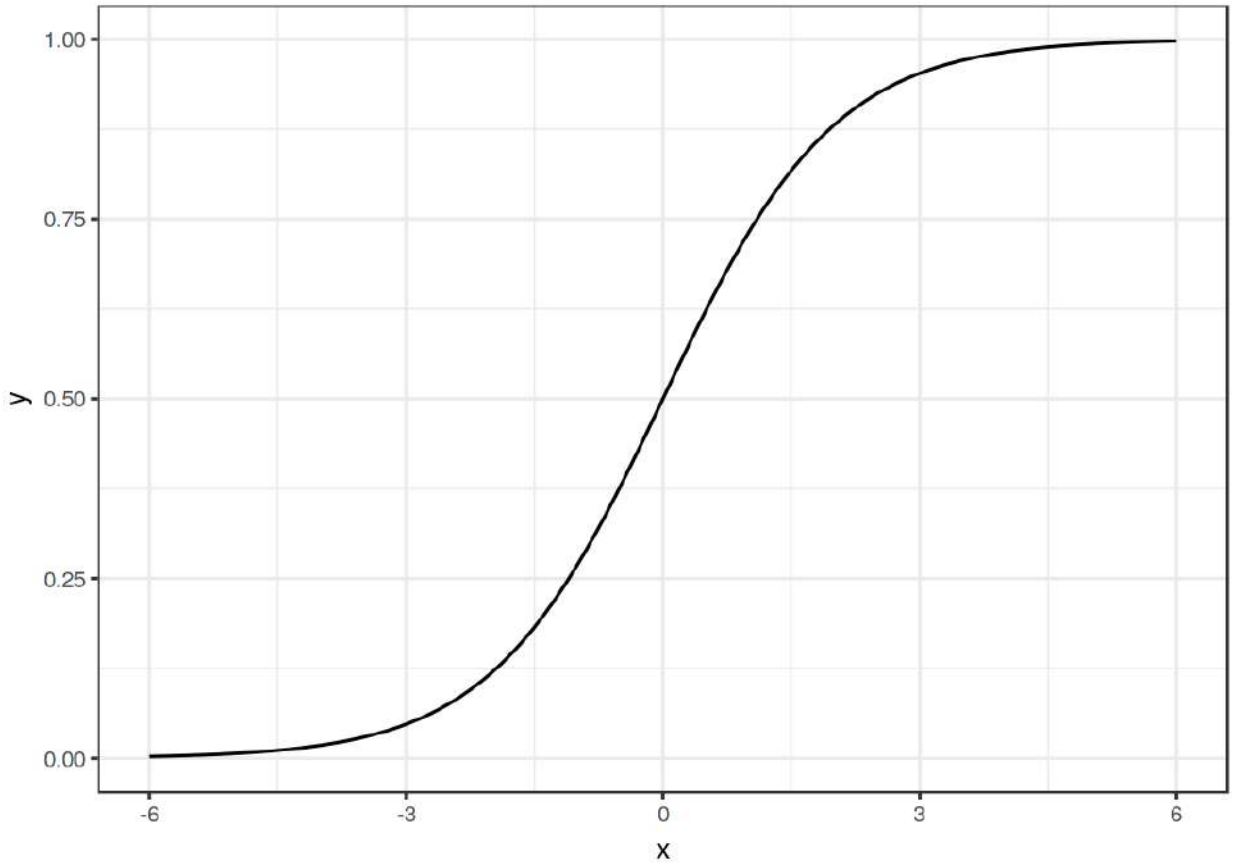
A linear model classifies tumors as malignant (1) or benign (0) given their size. The lines show the prediction of the linear model. For the data on the left, we can use 0.5 as classification threshold. After introducing a few more malignant tumor cases, the regression line shifts and a threshold of 0.5 no longer separates the classes. Points are slightly jittered to reduce over-plotting.

Theory

A solution for classification is logistic regression. Instead of fitting a straight line or hyperplane, the logistic regression model uses the logistic function to squeeze the output of a linear equation between 0 and 1. The logistic function is defined as:

$$\text{logistic}(\eta) = \frac{1}{1 + exp(-\eta)}$$

And it looks like this:



The logistic function. It outputs numbers between 0 and 1. At input 0, it outputs 0.5.

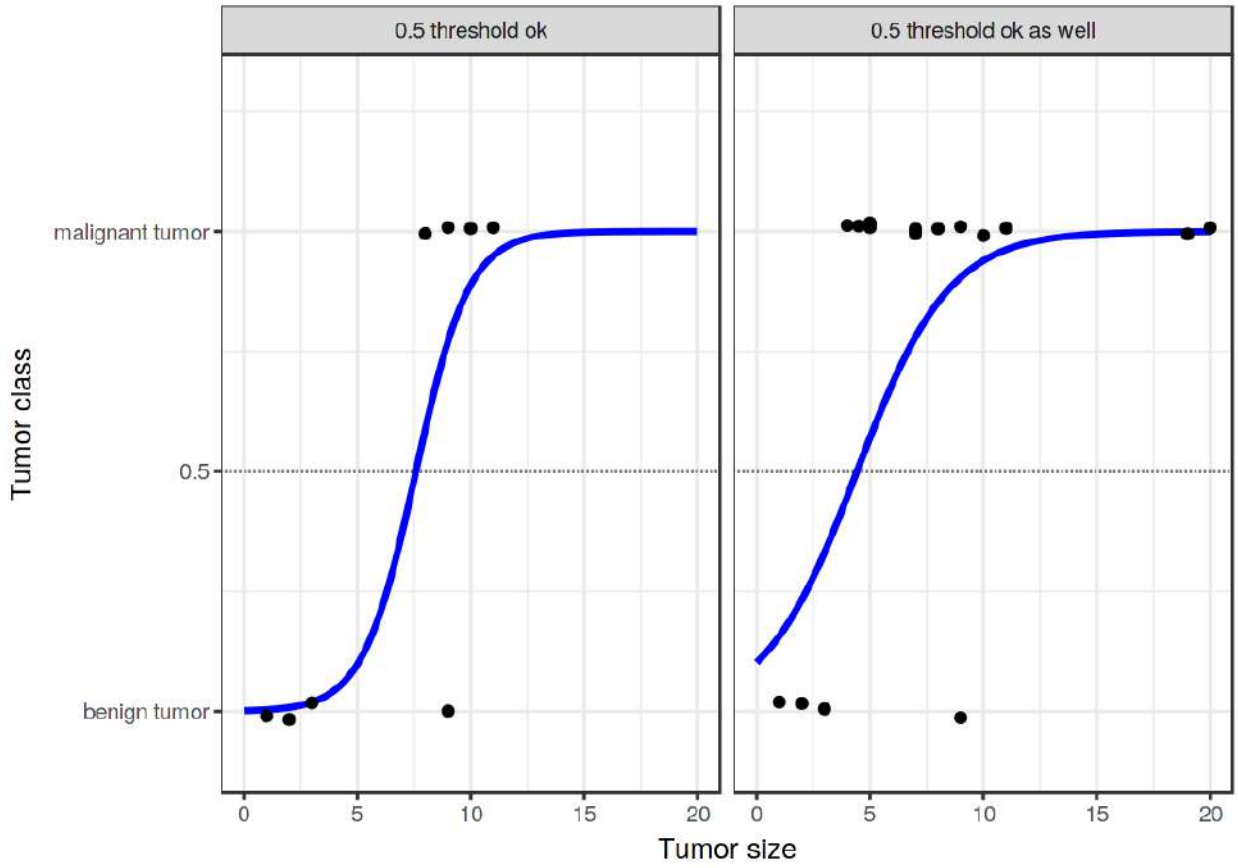
The step from linear regression to logistic regression is kind of straightforward. In the linear regression model, we have modelled the relationship between outcome and features with a linear equation:

$$\hat{y}^{(i)} = \beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_p x_p^{(i)}$$

For classification, we prefer probabilities between 0 and 1, so we wrap the right side of the equation into the logistic function. This forces the output to assume only values between 0 and 1.

$$P(y^{(i)} = 1) = \frac{1}{1 + exp(-(\beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_p x_p^{(i)}))}$$

Let us revisit the tumor size example again. But instead of the linear regression model, we use the logistic regression model:



The logistic regression model finds the correct decision boundary between malignant and benign depending on tumor size. The blue line is the logistic function shifted and squeezed to fit the data.

Classification works better with logistic regression and we can use 0.5 as a threshold in both cases. The inclusion of additional points does not really affect the estimated curve.

Interpretation

The interpretation of the weights in logistic regression differs from the interpretation of the weights in linear regression, since the outcome in logistic regression is a probability between 0 and 1. The weights do not influence the probability linearly any longer. The weighted sum is transformed by the logistic function to a probability. Therefore we need to reformulate the equation for the interpretation so that only the linear term is on the right side of the formula.

$$log\left(\frac{P(y=1)}{1-P(y=1)}\right) = log\left(\frac{P(y=1)}{P(y=0)}\right) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

We call the term in the log() function "odds" (probability of event divided by probability of no event) and wrapped in the logarithm it is called log odds.

This formula shows that the logistic regression model is a linear model for the log odds. Great! That does not sound helpful! With a little shuffling of the terms, you can figure out how the prediction changes when one of the features x_j is changed by 1 unit. To do this, we can first apply the exp() function to both sides of the equation:

$$\frac{P(y=1)}{1-P(y=1)} = odds = exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p\right)$$

Then we compare what happens when we increase one of the feature values by 1. But instead of looking at the difference, we look at the ratio of the two predictions:

$$\frac{odds_{x_j+1}}{odds} = \frac{exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_j(x_j+1) + \ldots + \beta_p x_p\right)}{exp\left(\beta_0 + \beta_1 x_1 + \ldots + \beta_j x_j + \ldots + \beta_p x_p\right)}$$

We apply the following rule:

$$\frac{exp(a)}{exp(b)} = exp(a-b)$$

And we remove many terms:

$$\frac{odds_{x_j+1}}{odds} = exp\left(\beta_j(x_j+1) - \beta_j x_j\right) = exp\left(\beta_j\right)$$

In the end, we have something as simple as exp() of a feature weight. A change in a feature by one unit changes the odds ratio (multiplicative) by a factor of exp(β_j). We could also interpret it this way: A change in x_j by one unit increases the log odds ratio by the value of the corresponding weight. Most people interpret the odds ratio because thinking about the log() of something is known to be hard on the brain. Interpreting the odds ratio already requires some getting used to. For example, if you have odds of 2, it means that the probability for y=1 is twice as high as y=0. If you have a weight (= log odds ratio) of 0.7, then increasing the respective feature by one unit multiplies the odds by exp(0.7) (approximately 2) and the odds change to 4. But usually you do not deal with the odds and interpret the weights only as the odds ratios. Because for actually calculating the odds you would need to set a value for each feature, which only makes sense if you want to look at one specific instance of your dataset.

These are the interpretations for the logistic regression model with different feature types:

- Numerical feature: If you increase the value of feature x_j by one unit, the estimated odds change by a factor of exp(β_j)
- Binary categorical feature: One of the two values of the feature is the reference category (in some languages, the one encoded in 0). Changing the feature x_j from the reference category to the other category changes the estimated odds by a factor of exp(β_j).
- Categorical feature with more than two categories: One solution to deal with multiple categories is one-hot-encoding, meaning that each category has its own column. You only need

L-1 columns for a categorical feature with L categories, otherwise it is over-parameterized. The L-th category is then the reference category. You can use any other encoding that can be used in linear regression. The interpretation for each category then is equivalent to the interpretation of binary features.

- Intercept β_0: When all numerical features are zero and the categorical features are at the reference category, the estimated odds are $\exp(\beta_0)$. The interpretation of the intercept weight is usually not relevant.

Example

We use the logistic regression model to predict cervical cancer based on some risk factors. The following table shows the estimate weights, the associated odds ratios, and the standard error of the estimates.

	Weight	Odds ratio	Std. Error
Intercept	2.91	18.36	0.32
Hormonal contraceptives y/n	0.12	1.12	0.30
Smokes y/n	-0.26	0.77	0.37
Num. of pregnancies	-0.04	0.96	0.10
Num. of diagnosed STDs	-0.82	0.44	0.33
Intrauterine device y/n	-0.62	0.54	0.40

Interpretation of a numerical feature ("Num. of diagnosed STDs"): An increase in the number of diagnosed STDs (sexually transmitted diseases) changes (decreases) the odds of cancer vs. no cancer by a factor of 0.44, when all other features remain the same. Keep in mind that correlation does not imply causation. No recommendation here to get STDs.

Interpretation of a categorical feature ("Hormonal contraceptives y/n"): For women using hormonal contraceptives, the odds for cancer vs. no cancer are by a factor of 1.12 higher, compared to women without hormonal contraceptives, given all other features stay the same.

Like in the linear model, the interpretations always come with the clause that 'all other features stay the same'.

Advantages and Disadvantages

Many of the pros and cons of the linear regression model also apply to the logistic regression model. Logistic regression has been widely used by many different people, but it struggles with its restrictive expressiveness (e.g. interactions must be added manually) and other models may have better predictive performance.

Another disadvantage of the logistic regression model is that the interpretation is more difficult because the interpretation of the weights is multiplicative and not additive.

Logistic regression can suffer from **complete separation**. If there is a feature that would perfectly separate the two classes, the logistic regression model can no longer be trained. This is because

the weight for that feature would not converge, because the optimal weight would be infinite. This is really a bit unfortunate, because such a feature is really useful. But you do not need machine learning if you have a simple rule that separates both classes. The problem of complete separation can be solved by introducing penalization of the weights or defining a prior probability distribution of weights.

On the good side, the logistic regression model is not only a classification model, but also gives you probabilities. This is a big advantage over models that can only provide the final classification. Knowing that an instance has a 99% probability for a class compared to 51% makes a big difference.

Logistic regression can also be extended from binary classification to multi-class classification. Then it is called Multinomial Regression.

Software

I used the `glm` function in R for all examples. You can find logistic regression in any programming language that can be used for performing data analysis, such as Python, Java, Stata, Matlab, …

GLM, GAM and more

The biggest strength but also the biggest weakness of the linear regression model is that the prediction is modeled as a weighted sum of the features. In addition, the linear model comes with many other assumptions. The bad news is (well, not really news) that all those assumptions are often violated in reality: The outcome given the features might have a non-Gaussian distribution, the features might interact and the relationship between the features and the outcome might be nonlinear. The good news is that the statistics community has developed a variety of modifications that transform the linear regression model from a simple blade into a Swiss knife.

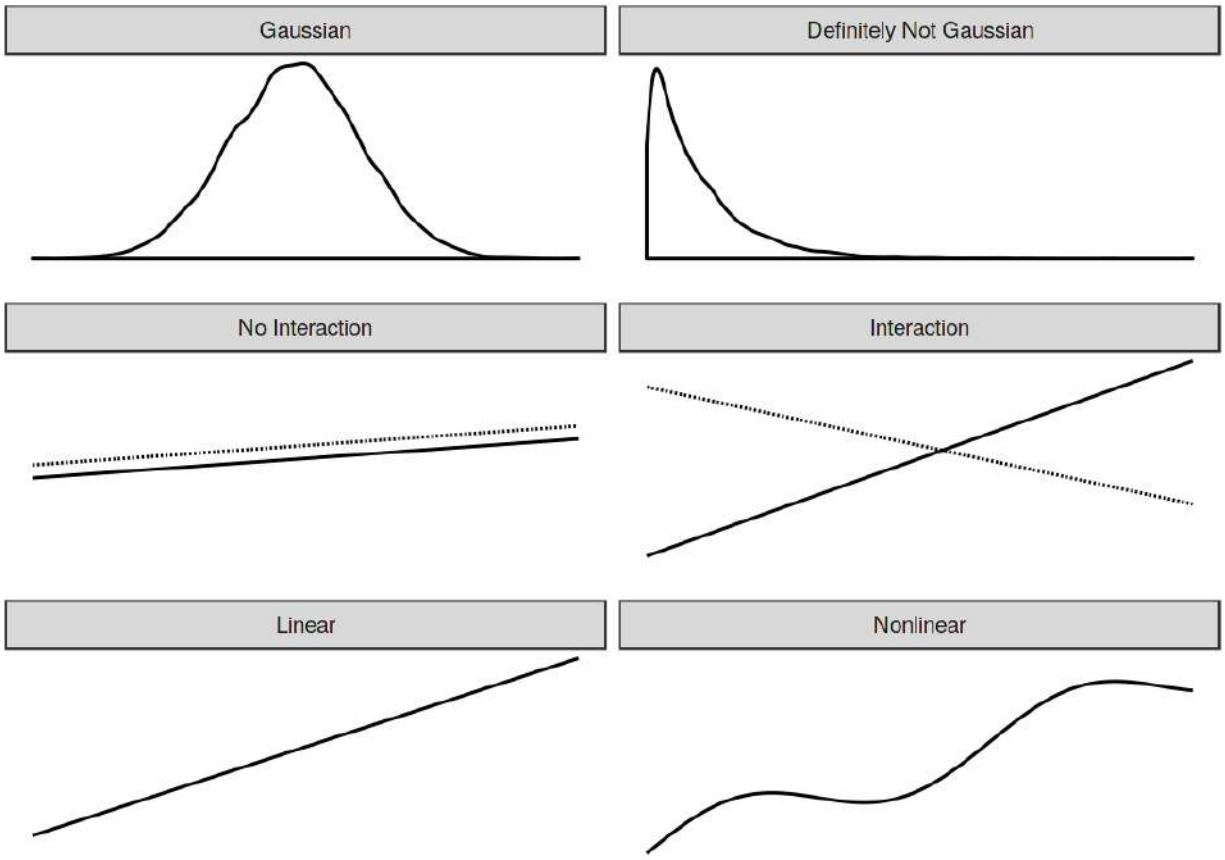
This chapter is definitely not your definite guide to extending linear models. Rather, it serves as an overview of extensions such as Generalized Linear Models (GLMs) and Generalized Additive Models (GAMs) and gives you a little intuition. After reading, you should have a solid overview of how to extend linear models. If you want to learn more about the linear regression model first, I suggest you read the chapter on linear regression models, if you have not already.

Let us remember the formula of a linear regression model:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \epsilon$$

The linear regression model assumes that the outcome y of an instance can be expressed by a weighted sum of its p features with an individual error ϵ that follows a Gaussian distribution. By forcing the data into this corset of a formula, we obtain a lot of model interpretability. The feature effects are additive, meaning no interactions, and the relationship is linear, meaning an increase of a feature by one unit can be directly translated into an increase/decrease of the predicted outcome. The linear model allows us to compress the relationship between a feature and the expected outcome into a single number, namely the estimated weight.

But a simple weighted sum is too restrictive for many real world prediction problems. In this chapter we will learn about three problems of the classical linear regression model and how to solve them. There are many more problems with possibly violated assumptions, but we will focus on the three shown in the following figure:



Three assumptions of the linear model (left side): Gaussian distribution of the outcome given the features, additivity (= no interactions) and linear relationship. Reality usually does not adhere to those assumptions (right side): Outcomes might have non-Gaussian distributions, features might interact and the relationship might be nonlinear.

There is a solution to all these problems:

Problem: The target outcome y given the features does not follow a Gaussian distribution.
Example: Suppose I want to predict how many minutes I will ride my bike on a given day. As features I have the type of day, the weather and so on. If I use a linear model, it could predict negative minutes because it assumes a Gaussian distribution which does not stop at 0 minutes. Also if I want to predict probabilities with a linear model, I can get probabilities that are negative or greater than 1.
Solution: Generalized Linear Models (GLMs).

Problem: The features interact.
Example: On average, light rain has a slight negative effect on my desire to go cycling. But in summer, during rush hour, I welcome rain, because then all the fair-weather cyclists stay at home and I have the bicycle paths for myself! This is an interaction between time and weather that cannot be captured by a purely additive model.
Solution: Adding interactions manually.

Problem: The true relationship between the features and y is not linear.
Example: Between 0 and 25 degrees Celsius, the influence of the temperature on my desire to ride

a bike could be linear, which means that an increase from 0 to 1 degree causes the same increase in cycling desire as an increase from 20 to 21. But at higher temperatures my motivation to cycle levels off and even decreases - I do not like to bike when it is too hot.
Solutions: Generalized Additive Models (GAMs); transformation of features.

The solutions to these three problems are presented in this chapter. Many further extensions of the linear model are omitted. If I attempted to cover everything here, the chapter would quickly turn into a book within a book about a topic that is already covered in many other books. But since you are already here, I have made a little problem plus solution overview for linear model extensions, which you can find at the end of the chapter. The name of the solution is meant to serve as a starting point for a search.

Non-Gaussian Outcomes - GLMs

The linear regression model assumes that the outcome given the input features follows a Gaussian distribution. This assumption excludes many cases: The outcome can also be a category (cancer vs. healthy), a count (number of children), the time to the occurrence of an event (time to failure of a machine) or a very skewed outcome with a few very high values (household income). The linear regression model can be extended to model all these types of outcomes. This extension is called **Generalized Linear Models** or **GLMs** for short. Throughout this chapter, I will use the name GLM for both the general framework and for particular models from that framework. The core concept of any GLM is: Keep the weighted sum of the features, but allow non-Gaussian outcome distributions and connect the expected mean of this distribution and the weighted sum through a possibly nonlinear function. For example, the logistic regression model assumes a Bernoulli distribution for the outcome and links the expected mean and the weighted sum using the logistic function.

The GLM mathematically links the weighted sum of the features with the mean value of the assumed distribution using the link function g, which can be chosen flexibly depending on the type of outcome.

$$g(E_Y(y|x)) = \beta_0 + \beta_1 x_1 + \ldots \beta_p x_p$$

GLMs consist of three components: The link function g, the weighted sum $X^T\beta$ (sometimes called linear predictor) and a probability distribution from the exponential family that defines E_Y.

The exponential family is a set of distributions that can be written with the same (parameterized) formula that includes an exponent, the mean and variance of the distribution and some other parameters. I will not go into the mathematical details because this is a very big universe of its own that I do not want to enter. Wikipedia has a neat list of distributions from the exponential family[39]. Any distribution from this list can be chosen for your GLM. Based on the type of the outcome you want to predict, choose a suitable distribution. Is the outcome a count of something (e.g. number of children living in a household)? Then the Poisson distribution could be a good choice. Is the outcome

[39] https://en.wikipedia.org/wiki/Exponential_family#Table_of_distributions

always positive (e.g. time between two events)? Then the exponential distribution could be a good choice.

Let us consider the classic linear model as a special case of a GLM. The link function for the Gaussian distribution in the classic linear model is simply the identity function. The Gaussian distribution is parameterized by the mean and the variance parameters. The mean describes the value that we expect on average and the variance describes how much the values vary around this mean. In the linear model, the link function links the weighted sum of the features to the mean of the Gaussian distribution.

Under the GLM framework, this concept generalizes to any distribution (from the exponential family) and arbitrary link functions. If y is a count of something, such as the number of coffees someone drinks on a certain day, we could model it with a GLM with a Poisson distribution and the natural logarithm as the link function:

$$ln(E_Y(y|x) = x^T\beta$$

The logistic regression model is also a GLM that assumes a Bernoulli distribution and uses the logistic function as the link function. The mean of the binomial distribution used in logistic regression is the probability that y is 1.

$$x^T\beta = ln\left(\frac{E_Y(y|x)}{1-E_Y(y|x)}\right) = ln\left(\frac{P(y=1|x)}{1-P(y=1|x)}\right)$$

And if we solve this equation to have P(y=1) on one side, we get the logistic regression formula:

$$P(y=1) = \frac{1}{1+exp(-x^T\beta)}$$

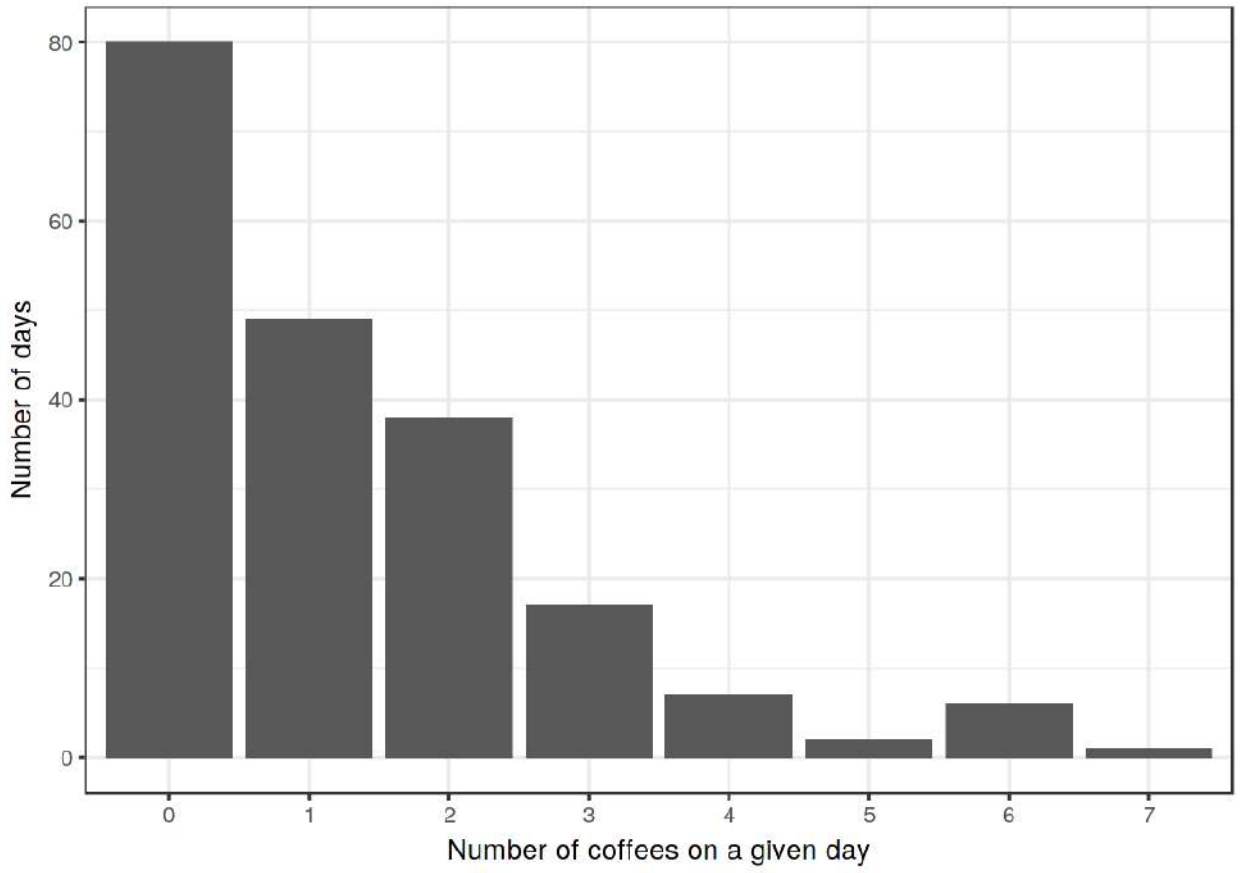
Each distribution from the exponential family has a canonical link function that can be derived mathematically from the distribution. The GLM framework makes it possible to choose the link function independently of the distribution. How to choose the right link function? There is no perfect recipe. You take into account knowledge about the distribution of your target, but also theoretical considerations and how well the model fits your actual data. For some distributions the canonical link function can lead to values that are invalid for that distribution. In the case of the exponential distribution, the canonical link function is the negative inverse, which can lead to negative predictions that are outside the domain of the exponential distribution. Since you can choose any link function, the simple solution is to choose another function that respects the domain of the distribution.

Examples

I have simulated a dataset on coffee drinking behavior to highlight the need for GLMs. Suppose you have collected data about your daily coffee drinking behavior. If you do not like coffee, pretend it is about tea. Along with number of cups, you record your current stress level on a scale of 1 to 10,

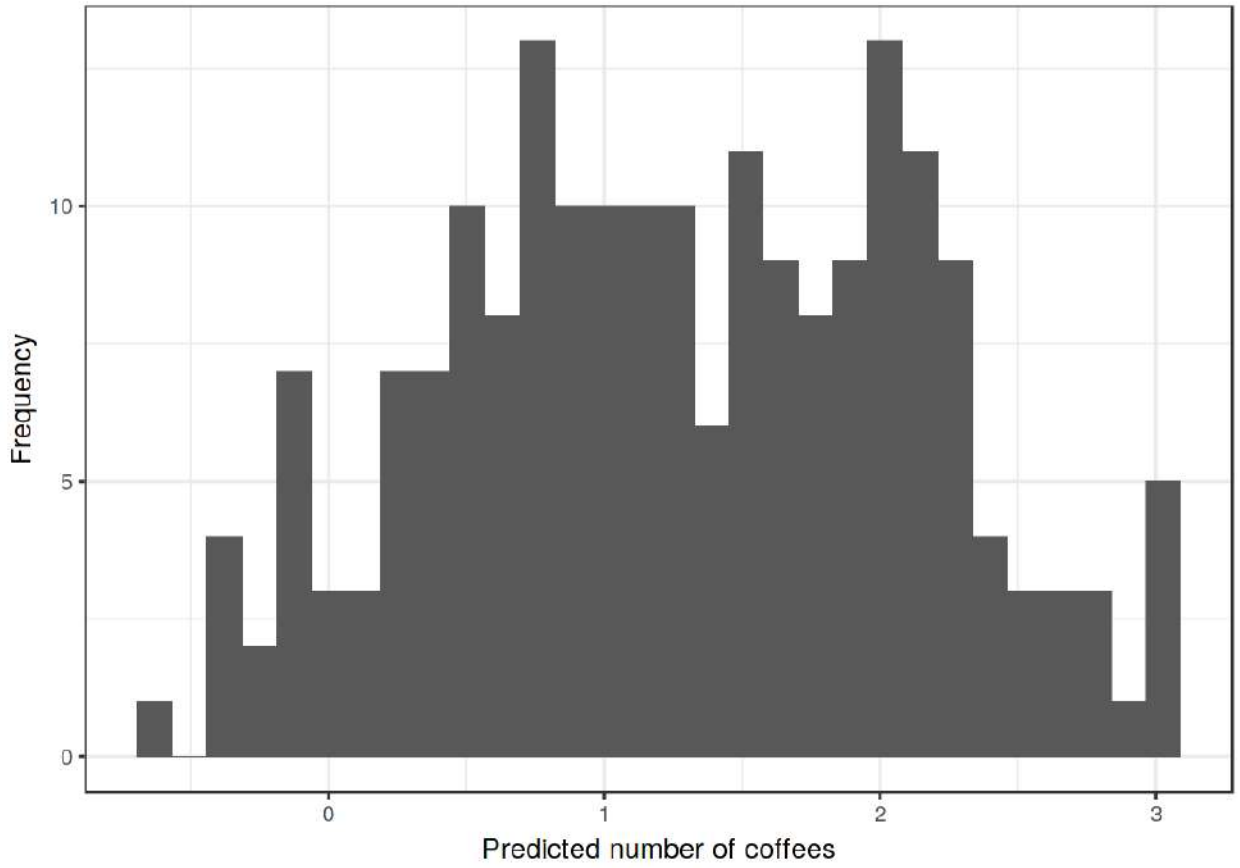
how well you slept the night before on a scale of 1 to 10 and whether you had to work on that day. The goal is to predict the number of coffees given the features stress, sleep and work. I simulated data for 200 days. Stress and sleep were drawn uniformly between 1 and 10 and work yes/no was drawn with a 50/50 chance (what a life!). For each day, the number of coffees was then drawn from a Poisson distribution, modelling the intensity λ (which is also the expected value of the Poisson distribution) as a function of the features sleep, stress and work. You can guess where this story will lead: *"Hey, let us model this data with a linear model ... Oh it does not work ... Let us try a GLM with Poisson distribution ... SURPRISE! Now it works!"*. I hope I did not spoil the story too much for you.

Let us look at the distribution of the target variable, the number of coffees on a given day:



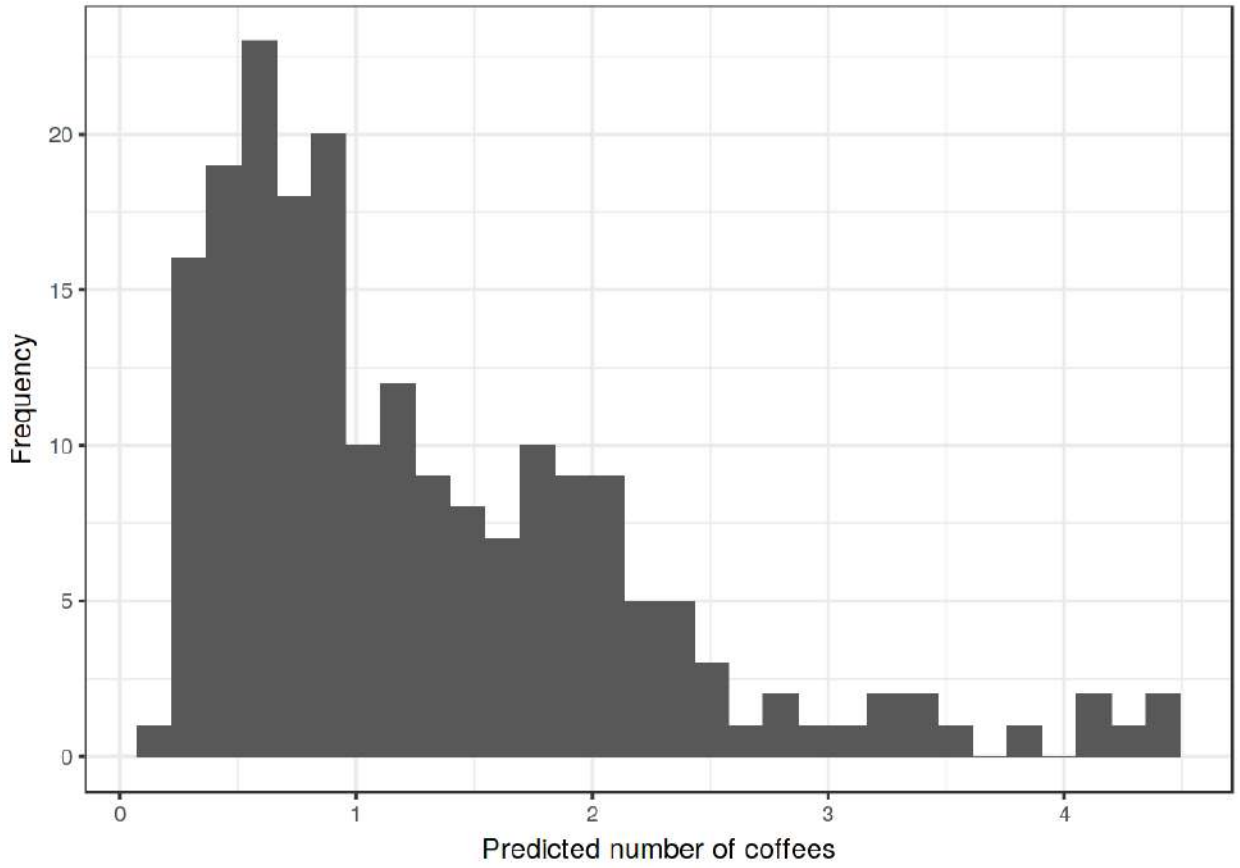
Simulated distribution of number of daily coffees for 200 days.

On 80 of the 200 days you had no coffee at all and on the most extreme day you had 7. Let us naively use a linear model to predict the number of coffees using sleep level, stress level and work yes/no as features. What can go wrong when we falsely assume a Gaussian distribution? A wrong assumption can invalidate the estimates, especially the confidence intervals of the weights. A more obvious problem is that the predictions do not match the "allowed" domain of the true outcome, as the following figure shows.



Predicted number of coffees dependent on stress, sleep and work. The linear model predicts negative values.

The linear model does not make sense, because it predicts negative number of coffees. This problem can be solved with Generalized Linear Models (GLMs). We can change the link function and the assumed distribution. One possibility is to keep the Gaussian distribution and use a link function that always leads to positive predictions such as the log-link (the inverse is the exp-function) instead of the identity function. Even better: We choose a distribution that corresponds to the data generating process and an appropriate link function. Since the outcome is a count, the Poisson distribution is a natural choice, along with the logarithm as link function. In this case, the data was even generated with the Poisson distribution, so the Poisson GLM is the perfect choice. The fitted Poisson GLM leads to the following distribution of predicted values:



Predicted number of coffees dependent on stress, sleep and work. The GLM with Poisson assumption and log link is an appropriate model for this dataset.

No negative amounts of coffees, looks much better now.

Interpretation of GLM weights

The assumed distribution together with the link function determines how the estimated feature weights are interpreted. In the coffee count example, I used a GLM with Poisson distribution and log link, which implies the following relationship between the features and the expected outcome.

$$ln(E(\text{coffees}|\text{stress}, \text{sleep}, \text{work})) = \beta_0 + \beta_{\text{stress}}x_{\text{stress}} + \beta_{\text{sleep}}x_{\text{sleep}} + \beta_{\text{work}}x_{\text{work}}$$

To interpret the weights we invert the link function so that we can interpret the effect of the features on the expected outcome and not on the logarithm of the expected outcome.

$$E(\text{coffees}|\text{stress}, \text{sleep}, \text{work}) = exp(\beta_0 + \beta_{\text{stress}}x_{\text{stress}} + \beta_{\text{sleep}}x_{\text{sleep}} + \beta_{\text{work}}x_{\text{work}})$$

Since all the weights are in the exponential function, the effect interpretation is not additive, but multiplicative, because exp(a + b) is exp(a) times exp(b). The last ingredient for the interpretation

is the actual weights of the toy example. The following table lists the estimated weights and exp(weights) together with the 95% confidence interval:

	weight	exp(weight) [2.5%, 97.5%]
(Intercept)	-0.12	0.89 [0.56, 1.38]
stress	0.11	1.11 [1.06, 1.17]
sleep	-0.16	0.85 [0.81, 0.89]
workYES	0.88	2.42 [1.87, 3.16]

Increasing the stress level by one point multiplies the expected number of coffees by the factor 1.11. Increasing the sleep quality by one point multiplies the expected number of coffees by the factor 0.85. The predicted number of coffees on a work day is on average 2.42 times the number of coffees on a day off. In summary, the more stress, the less sleep and the more work, the more coffee is consumed.

In this section you learned a little about Generalized Linear Models that are useful when the target does not follow a Gaussian distribution. Next, we look at how to integrate interactions between two features into the linear regression model.

Interactions

The linear regression model assumes that the effect of one feature is the same regardless of the values of the other features (= no interactions). But often there are interactions in the data. To predict the number of bicycles rented, there may be an interaction between temperature and whether it is a working day or not. Perhaps, when people have to work, the temperature does not influence the number of rented bikes much, because people will ride the rented bike to work no matter what happens. On days off, many people ride for pleasure, but only when it is warm enough. When it comes to rental bicycles, you might expect an interaction between temperature and working day.

How can we get the linear model to include interactions? Before you fit the linear model, add a column to the feature matrix that represents the interaction between the features and fit the model as usual. The solution is elegant in a way, since it does not require any change of the linear model, only additional columns in the data. In the working day and temperature example, we would add a new feature that has zeros for no-work days, otherwise it has the value of the temperature feature, assuming that working day is the reference category. Suppose our data looks like this:

work	temp
Y	25
N	12
N	30
Y	5

The data matrix used by the linear model looks slightly different. The following table shows what the data prepared for the model looks like if we do not specify any interactions. Normally, this transformation is performed automatically by any statistical software.

Intercept	workY	temp
1	1	25
1	0	12
1	0	30
1	1	5

The first column is the intercept term. The second column encodes the categorical feature, with 0 for the reference category and 1 for the other. The third column contains the temperature.

If we want the linear model to consider the interaction between temperature and the workingday feature, we have to add a column for the interaction:

Intercept	workY	temp	workY.temp
1	1	25	25
1	0	12	0
1	0	30	0
1	1	5	5

The new column "workY.temp" captures the interaction between the features working day (work) and temperature (temp). This new feature column is zero for an instance if the work feature is at the reference category ("N" for no working day), otherwise it assumes the values of the instances temperature feature. With this type of encoding, the linear model can learn a different linear effect of temperature for both types of days. This is the interaction effect between the two features. Without an interaction term, the combined effect of a categorical and a numerical feature can be described by a line that is vertically shifted for the different categories. If we include the interaction, we allow the effect of the numerical features (the slope) to have a different value in each category.

The interaction of two categorical features works similarly. We create additional features which represent combinations of categories. Here is some artificial data containing working day (work) and a categorical weather feature (wthr):

work	wthr
Y	Good
N	Bad
N	Ok
Y	Good

Next, we include interaction terms:

Intercept	workY	wthrGood	wthrOk	workY.wthrGood	workY.wthrOk
1	1	1	0	1	0
1	0	0	0	0	0
1	0	0	1	0	0
1	1	1	0	1	0

The first column serves to estimate the intercept. The second column is the encoded work feature. Columns three and four are for the weather feature, which requires two columns because you

need two weights to capture the effect for three categories, one of which is the reference category. The rest of the columns capture the interactions. For each category of both features (except for the reference categories), we create a new feature column that is 1 if both features have a certain category, otherwise 0.

For two numerical features, the interaction column is even easier to construct: We simply multiply both numerical features.

There are approaches to automatically detect and add interaction terms. One of them can be found in the RuleFit chapter. The RuleFit algorithm first mines interaction terms and then estimates a linear regression model including interactions.

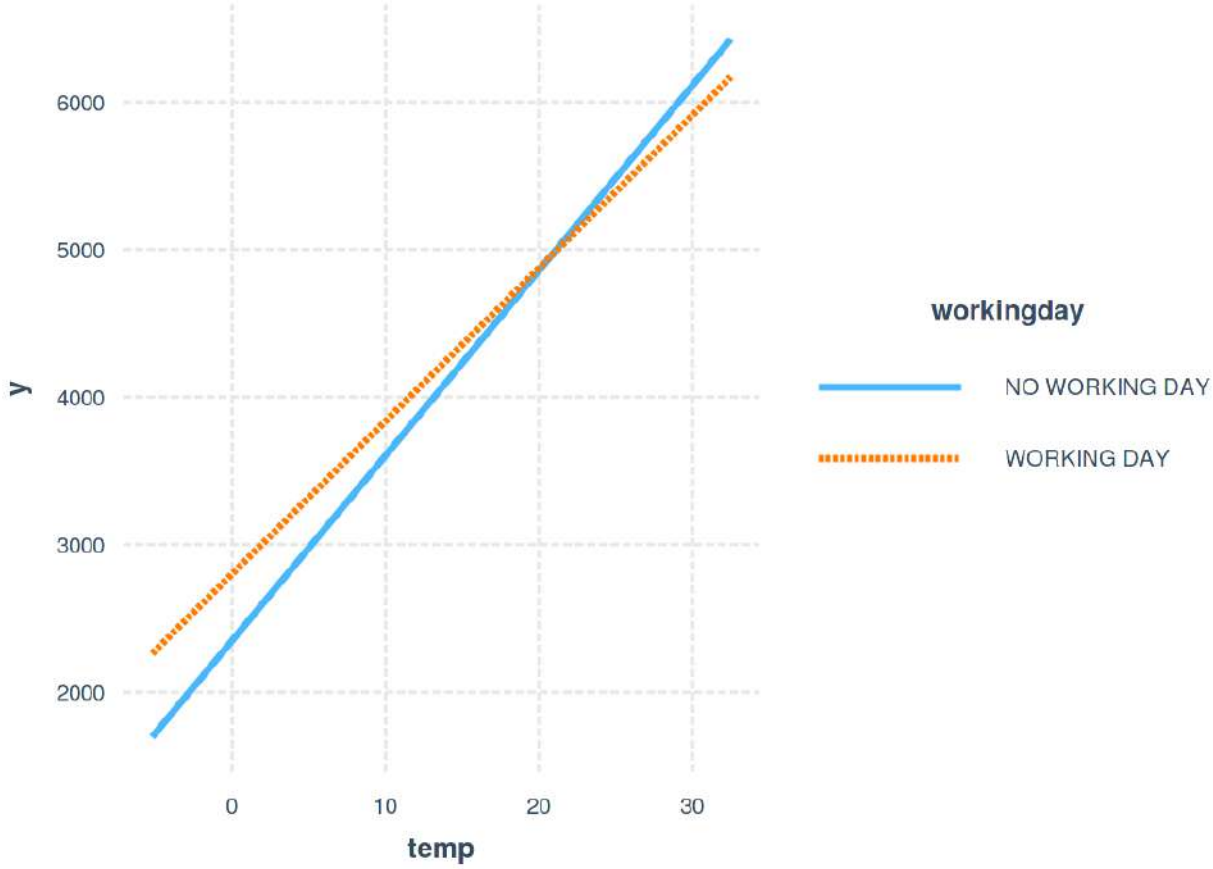
Example

Let us return to the bike rental prediction task which we have already modeled in the linear model chapter. This time, we additionally consider an interaction between the temperature and the working day feature. This results in the following estimated weights and confidence intervals.

	Weight	Std. Error	2.5%	97.5%
(Intercept)	2185.8	250.2	1694.6	2677.1
seasonSUMMER	893.8	121.8	654.7	1132.9
seasonFALL	137.1	161.0	-179.0	453.2
seasonWINTER	426.5	110.3	209.9	643.2
holidayHOLIDAY	-674.4	202.5	-1071.9	-276.9
workingdayWORKING DAY	451.9	141.7	173.7	730.1
weathersitMISTY	-382.1	87.2	-553.3	-211.0
weathersitRAIN/SNOW/STORM	-1898.2	222.7	-2335.4	-1461.0
temp	125.4	8.9	108.0	142.9
hum	-17.5	3.2	-23.7	-11.3
windspeed	-42.1	6.9	-55.5	-28.6
days_since_2011	4.9	0.2	4.6	5.3
workingdayWORKING DAY:temp	-21.8	8.1	-37.7	-5.9

The additional interaction effect is negative (-21.8) and differs significantly from zero, as shown by the 95% confidence interval, which does not include zero. By the way, the data are not iid, because days that are close to each other are not independent from each other. Confidence intervals might be misleading, just take it with a grain of salt. The interaction term changes the interpretation of the weights of the involved features. Does the temperature have a negative effect given it is a working day? The answer is no, even if the table suggests it to an untrained user. We cannot interpret the "workingdayWORKING DAY:temp" interaction weight in isolation, since the interpretation would be: "While leaving all other feature values unchanged, increasing the interaction effect of temperature for working day decreases the predicted number of bikes." But the interaction effect only adds to the main effect of the temperature. Suppose it is a working day and we want to know what would happen if the temperature were 1 degree warmer today. Then we need to sum both the weights for "temp" and "workingdayWORKING DAY:temp" to determine how much the estimate increases.

It is easier to understand the interaction visually. By introducing an interaction term between a categorical and a numerical feature, we get two slopes for the temperature instead of one. The temperature slope for days on which people do not have to work ('NO WORKING DAY') can be read directly from the table (125.4). The temperature slope for days on which people have to work ('WORKING DAY') is the sum of both temperature weights (125.4 -21.8 = 103.6). The intercept of the 'NO WORKING DAY'-line at temperature = 0 is determined by the intercept term of the linear model (2185.8). The intercept of the 'WORKING DAY'-line at temperature = 0 is determined by the intercept term + the effect of working day (2185.8 + 451.9 = 2637.7).



The effect (including interaction) of temperature and working day on the predicted number of bikes for a linear model. Effectively, we get two slopes for the temperature, one for each category of the working day feature.

Nonlinear Effects - GAMs

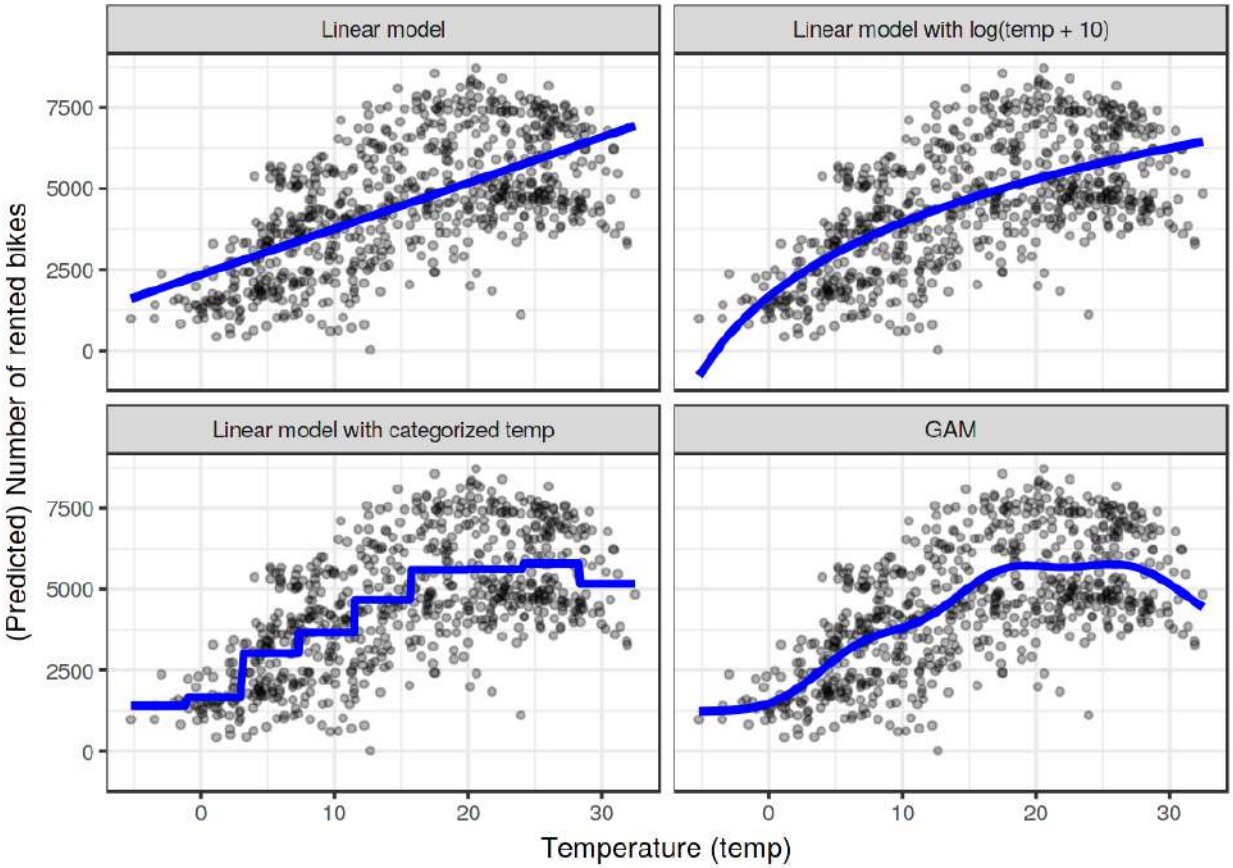
The world is not linear. Linearity in linear models means that no matter what value an instance has in a particular feature, increasing the value by one unit always has the same effect on the predicted outcome. Is it reasonable to assume that increasing the temperature by one degree at 10 degrees Celsius has the same effect on the number of rental bikes as increasing the temperature when it already has 40 degrees? Intuitively, one expects that increasing the temperature from 10 to 11 degrees Celsius has a positive effect on bicycle rentals and from 40 to 41 a negative effect, which is also the

case, as you will see, in many examples throughout the book. The temperature feature has a linear, positive effect on the number of rental bikes, but at some point it flattens out and even has a negative effect at high temperatures. The linear model does not care, it will dutifully find the best linear plane (by minimizing the Euclidean distance).

You can model nonlinear relationships using one of the following techniques:

- Simple transformation of the feature (e.g. logarithm)
- Categorization of the feature
- Generalized Additive Models (GAMs)

Before I go into the details of each method, let us start with an example that illustrates all three of them. I took the bike rental dataset and trained a linear model with only the temperature feature to predict the number of rental bikes. The following figure shows the estimated slope with: the standard linear model, a linear model with transformed temperature (logarithm), a linear model with temperature treated as categorical feature and using regression splines (GAM).



Predicting the number of rented bicycles using only the temperature feature. A linear model (top left) does not fit the data well. One solution is to transform the feature with e.g. the logarithm (top right), categorize it (bottom left), which is usually a bad decision, or use Generalized Additive Models that can automatically fit a smooth curve for temperature (bottom right).

Feature transformation

Often the logarithm of the feature is used as a transformation. Using the logarithm indicates that every 10-fold temperature increase has the same linear effect on the number of bikes, so changing from 1 degree Celsius to 10 degrees Celsius has the same effect as changing from 0.1 to 1 (sounds wrong). Other examples for feature transformations are the square root, the square function and the exponential function. Using a feature transformation means that you replace the column of this feature in the data with a function of the feature, such as the logarithm, and fit the linear model as usual. Some statistical programs also allow you to specify transformations in the call of the linear model. You can be creative when you transform the feature. The interpretation of the feature changes according to the selected transformation. If you use a log transformation, the interpretation in a linear model becomes: "If the logarithm of the feature is increased by one, the prediction is increased by the corresponding weight." When you use a GLM with a link function that is not the identity function, then the interpretation gets more complicated, because you have to incorporate both transformations into the interpretation (except when they cancel each other out, like log and exp, then the interpretation gets easier).

Feature categorization

Another possibility to achieve a nonlinear effect is to discretize the feature; turn it into a categorical feature. For example, you could cut the temperature feature into 20 intervals with the levels [-10, -5), [-5, 0), ... and so on. When you use the categorized temperature instead of the continuous temperature, the linear model would estimate a step function because each level gets its own estimate. The problem with this approach is that it needs more data, it is more likely to overfit and it is unclear how to discretize the feature meaningfully (equidistant intervals or quantiles? how many intervals?). I would only use discretization if there is a very strong case for it. For example, to make the model comparable to another study.

Generalized Additive Models (GAMs)

Why not 'simply' allow the (generalized) linear model to learn nonlinear relationships? That is the motivation behind GAMs. GAMs relax the restriction that the relationship must be a simple weighted sum, and instead assume that the outcome can be modeled by a sum of arbitrary functions of each feature. Mathematically, the relationship in a GAM looks like this:

$$g(E_Y(y|x)) = \beta_0 + f_1(x_1) + f_2(x_2) + \ldots + f_p(x_p)$$

The formula is similar to the GLM formula with the difference that the linear term $\beta_j x_j$ is replaced by a more flexible function $f_j(x_j)$. The core of a GAM is still a sum of feature effects, but you have the option to allow nonlinear relationships between some features and the output. Linear effects are also covered by the framework, because for features to be handled linearly, you can limit their $f_j(x_j)$ only to take the form of $x_j\beta_j$.

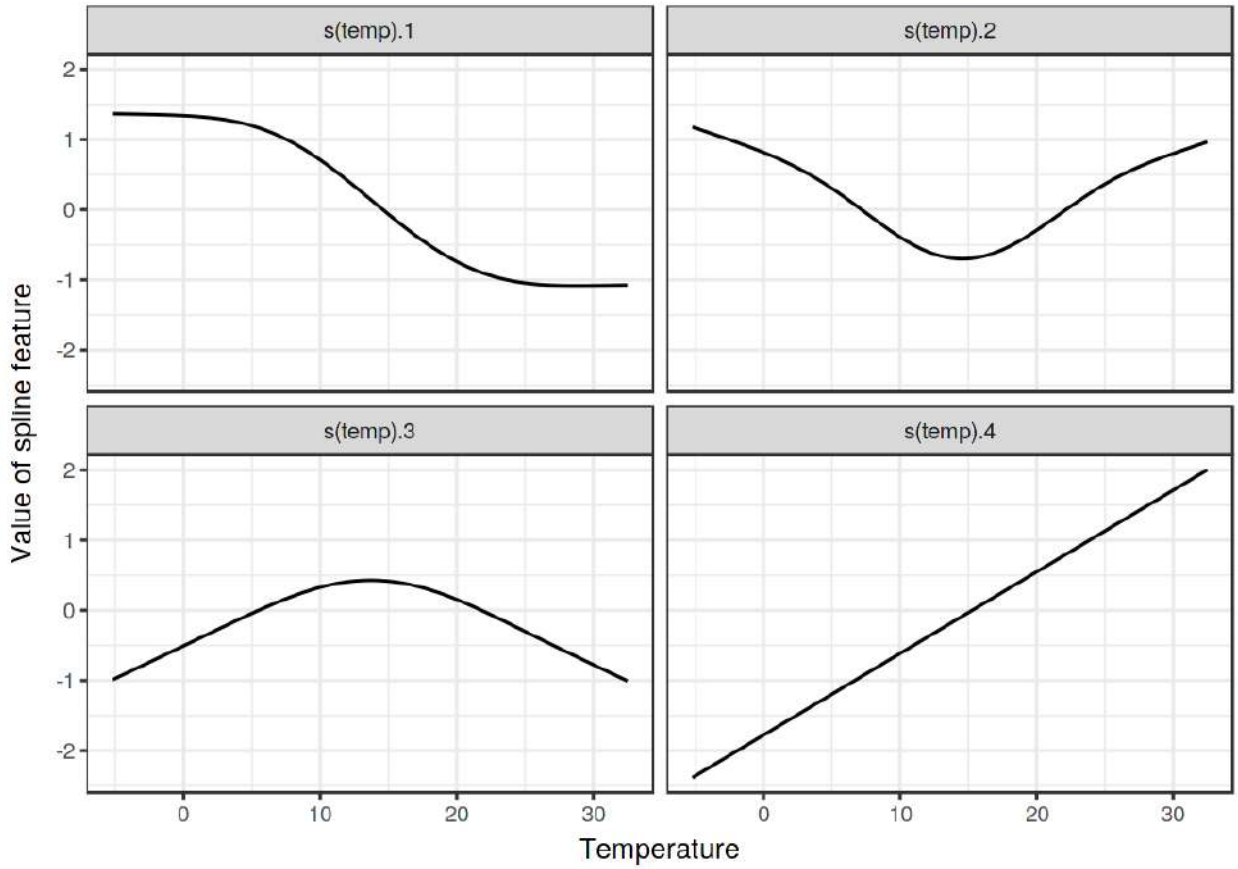
The big question is how to learn nonlinear functions. The answer is called "splines" or "spline functions". Splines are functions that can be combined in order to approximate arbitrary functions. A bit like stacking Lego bricks to build something more complex. There is a confusing number of

ways to define these spline functions. If you are interested in learning more about all the ways to define splines, I wish you good luck on your journey. I am not going to go into details here, I am just going to build an intuition. What personally helped me the most for understanding splines was to visualize the individual spline functions and to look into how the data matrix is modified. For example, to model the temperature with splines, we remove the temperature feature from the data and replace it with, say, 4 columns, each representing a spline function. Usually you would have more spline functions, I only reduced the number for illustration purposes. The value for each instance of these new spline features depends on the instances' temperature values. Together with all linear effects, the GAM then also estimates these spline weights. GAMs also introduce a penalty term for the weights to keep them close to zero. This effectively reduces the flexibility of the splines and reduces overfitting. A smoothness parameter that is commonly used to control the flexibility of the curve is then tuned via cross-validation. Ignoring the penalty term, nonlinear modeling with splines is fancy feature engineering.

In the example where we are predicting the number of bicycles with a GAM using only the temperature, the model feature matrix looks like this:

(Intercept)	s(temp).1	s(temp).2	s(temp).3	s(temp).4
1	0.93	-0.14	0.21	-0.83
1	0.83	-0.27	0.27	-0.72
1	1.32	0.71	-0.39	-1.63
1	1.32	0.70	-0.38	-1.61
1	1.29	0.58	-0.26	-1.47
1	1.32	0.68	-0.36	-1.59

Each row represents an individual instance from the data (one day). Each spline column contains the value of the spline function at the particular temperature values. The following figure shows how these spline functions look like:

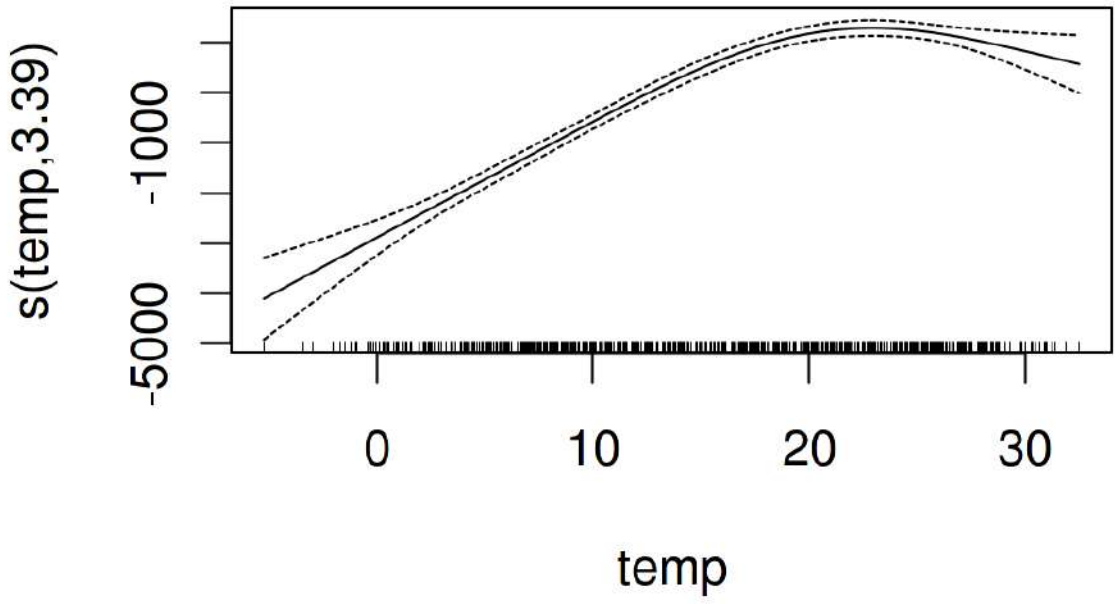


To smoothly model the temperature effect, we use 4 spline functions. Each temperature value is mapped to (here) 4 spline values. If an instance has a temperature of 30 ℃, the value for the first spline feature is -1, for the second 0.7, for the third -0.8 and for the 4th 1.7.

The GAM assigns weights to each temperature spline feature:

	weight
(Intercept)	4504.35
s(temp).1	-989.34
s(temp).2	740.08
s(temp).3	2309.84
s(temp).4	558.27

And the actual curve, which results from the sum of the spline functions weighted with the estimated weights, looks like this:



GAM feature effect of the temperature for predicting the number of rented bikes (temperature used as the only feature).

The interpretation of smooth effects requires a visual check of the fitted curve. Splines are usually centered around the mean prediction, so a point on the curve is the difference to the mean prediction. For example, at 0 degrees Celsius, the predicted number of bicycles is 3000 lower than the average prediction.

Advantages

All these extensions of the linear model are a bit of a universe in themselves. Whatever problems you face with linear models, **you will probably find an extension that fixes it**.

Most methods have been used for decades. For example, GAMs are almost 30 years old. Many researchers and practitioners from industry are very **experienced** with linear models and the methods are **accepted in many communities as status quo for modeling**.

In addition to making predictions, you can use the models to **do inference**, draw conclusions about the data – given the model assumptions are not violated. You get confidence intervals for weights, significance tests, prediction intervals and much more.

Statistical software usually has really good interfaces to fit GLMs, GAMs and more special linear models.

The opacity of many machine learning models comes from 1) a lack of sparseness, which means that many features are used, 2) features that are treated in a nonlinear fashion, which means you need more than a single weight to describe the effect, and 3) the modeling of interactions between the features. Assuming that linear models are highly interpretable but often underfit reality, the extensions described in this chapter offer a good way to achieve a **smooth transition to more flexible models**, while preserving some of the interpretability.

Disadvantages

As advantage I have said that linear models live in their own universe. The sheer **number of ways you can extend the simple linear model is overwhelming**, not just for beginners. Actually, there are multiple parallel universes, because many communities of researchers and practitioners have their own names for methods that do more or less the same thing, which can be very confusing.

Most modifications of the linear model make the model **less interpretable**. Any link function (in a GLM) that is not the identity function complicates the interpretation; interactions also complicate the interpretation; nonlinear feature effects are either less intuitive (like the log transformation) or can no longer be summarized by a single number (e.g. spline functions).

GLMs, GAMs and so on **rely on assumptions** about the data generating process. If those are violated, the interpretation of the weights is no longer valid.

The performance of tree-based ensembles like the random forest or gradient tree boosting is in many cases better than the most sophisticated linear models. This is partly my own experience and partly observations from the winning models on platforms like kaggle.com.

Software

All examples in this chapter were created using the R language. For GAMs, the `gam` package was used, but there are many others. R has an incredible number of packages to extend linear regression models. Unsurpassed by any other analytics language, R is home to every conceivable extension of the linear regression model extension. You will find implementations of e.g. GAMs in Python (such as pyGAM[40]), but theses implementation are not as mature.

Further Extensions

As promised, here is a list of problems you might encounter with linear models, along with the name of a solution for this problem that you can copy and paste into your favorite search engine.

[40] https://github.com/dswah/pyGAM

My data violates the assumption of being independent and identically distributed (iid).
For example, repeated measurements on the same patient.
Search for **mixed models** or **generalized estimating equations**.

My model has heteroscedastic errors.
For example, when predicting the value of a house, the model errors are usually higher in expensive houses, which violates the homoscedasticity of the linear model.
Search for **robust regression**.

I have outliers that strongly influence my model.
Search for **robust regression**.

I want to predict the time until an event occurs.
Time-to-event data usually comes with censored measurements, which means that for some instances there was not enough time to observe the event. For example, a company wants to predict the failure of its ice machines, but only has data for two years. Some machines are still intact after two years, but might fail later.
Search for **parametric survival models**, **cox regression**, **survival analysis**.

My outcome to predict is a category.
If the outcome has two categories use a logistic regression model, which models the probability for the categories.
If you have more categories, search for **multinomial regression**.
Logistic regression and multinomial regression are both GLMs.

I want to predict ordered categories.
For example school grades.
Search for **proportional odds model**.

My outcome is a count (like number of children in a family).
Search for **Poisson regression**.
The Poisson model is also a GLM. You might also have the problem that the count value of 0 is very frequent.
Search for **zero-inflated Poisson regression**, **hurdle model**.

I am not sure what features need to be included in the model to draw correct causal conclusions.
For example, I want to know the effect of a drug on the blood pressure. The drug has a direct effect on some blood value and this blood value affects the outcome. Should I include the blood value into the regression model?
Search for **causal inference**, **mediation analysis**.

I have missing data.
Search for **multiple imputation**.

I want to integrate prior knowledge into my models.
Search for **Bayesian inference**.

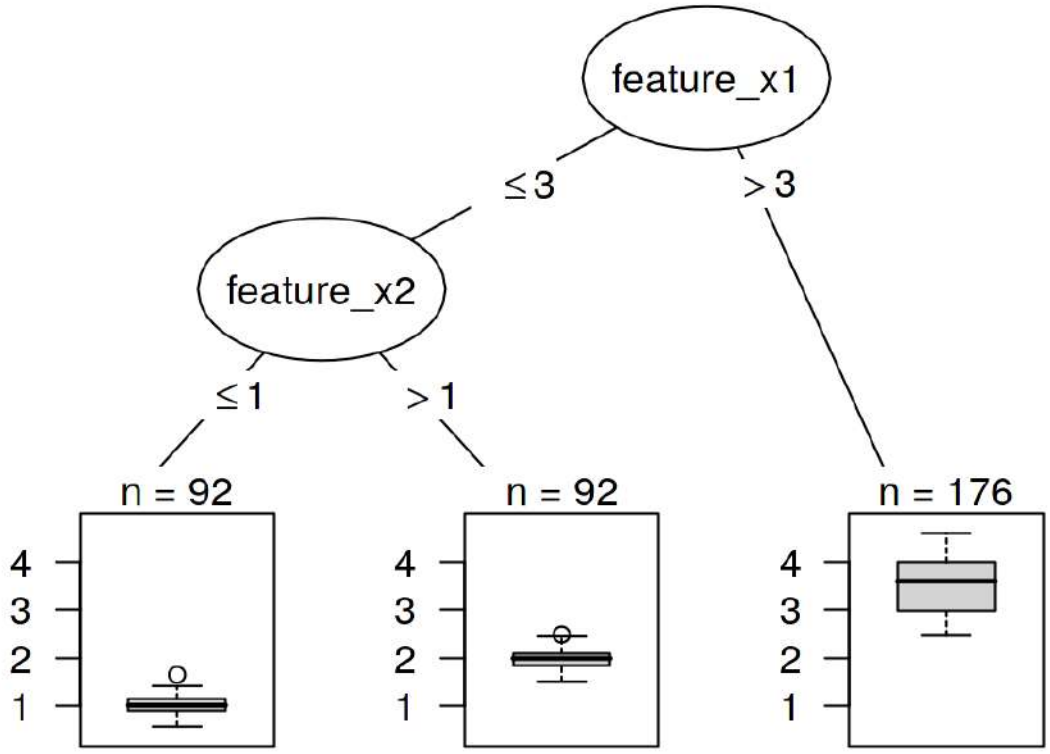
I am feeling a bit down lately.
Search for "**Amazon Alexa Gone Wild!!! Full version from beginning to end**".

Decision Tree

Linear regression and logistic regression models fail in situations where the relationship between features and outcome is nonlinear or where features interact with each other. Time to shine for the decision tree! Tree based models split the data multiple times according to certain cutoff values in the features. Through splitting, different subsets of the dataset are created, with each instance belonging to one subset. The final subsets are called terminal or leaf nodes and the intermediate subsets are called internal nodes or split nodes. To predict the outcome in each leaf node, the average outcome of the training data in this node is used. Trees can be used for classification and regression.

There are various algorithms that can grow a tree. They differ in the possible structure of the tree (e.g. number of splits per node), the criteria how to find the splits, when to stop splitting and how to estimate the simple models within the leaf nodes. The classification and regression trees (CART) algorithm is probably the most popular algorithm for tree induction. We will focus on CART, but the interpretation is similar for most other tree types. I recommend the book 'The Elements of Statistical Learning' (Friedman, Hastie and Tibshirani 2009)[41] for a more detailed introduction to CART.

[41] Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. "The elements of statistical learning". www.web.stanford.edu/~hastie/ElemStatLearn/ (2009).



Decision tree with artificial data. Instances with a value greater than 3 for feature x1 end up in node 5. All other instances are assigned to node 3 or node 4, depending on whether values of feature x2 exceed 1.

The following formula describes the relationship between the outcome y and features x.

$$\hat{y} = \hat{f}(x) = \sum_{m=1}^{M} c_m I\{x \in R_m\}$$

Each instance falls into exactly one leaf node (=subset R_m). $I_{\{x \in R_m\}}$ is the identity function that returns 1 if x is in the subset R_m and 0 otherwise. If an instance falls into a leaf node R_l, the predicted outcome is $\hat{y} = c_l$, where c_l is the average of all training instances in leaf node R_l.

But where do the subsets come from? This is quite simple: CART takes a feature and determines which cut-off point minimizes the variance of y for a regression task or the Gini index of the class distribution of y for classification tasks. The variance tells us how much the y values in a node are spread around their mean value. The Gini index tells us how "impure" a node is, e.g. if all classes have the same frequency, the node is impure, if only one class is present, it is maximally pure. Variance and Gini index are minimized when the data points in the nodes have very similar values for y. As a consequence, the best cut-off point makes the two resulting subsets as different as possible with respect to the target outcome. For categorical features, the algorithm tries to create subsets

by trying different groupings of categories. After the best cutoff per feature has been determined, the algorithm selects the feature for splitting that would result in the best partition in terms of the variance or Gini index and adds this split to the tree. The algorithm continues this search-and-split recursively in both new nodes until a stop criterion is reached. Possible criteria are: A minimum number of instances that have to be in a node before the split, or the minimum number of instances that have to be in a terminal node.

Interpretation

The interpretation is simple: Starting from the root node, you go to the next nodes and the edges tell you which subsets you are looking at. Once you reach the leaf node, the node tells you the predicted outcome. All the edges are connected by 'AND'.

Template: If feature x is [smaller/bigger] than threshold c AND ... then the predicted outcome is the mean value of y of the instances in that node.

Feature importance

The overall importance of a feature in a decision tree can be computed in the following way: Go through all the splits for which the feature was used and measure how much it has reduced the variance or Gini index compared to the parent node. The sum of all importances is scaled to 100. This means that each importance can be interpreted as share of the overall model importance.

Tree decomposition

Individual predictions of a decision tree can be explained by decomposing the decision path into one component per feature. We can track a decision through the tree and explain a prediction by the contributions added at each decision node.

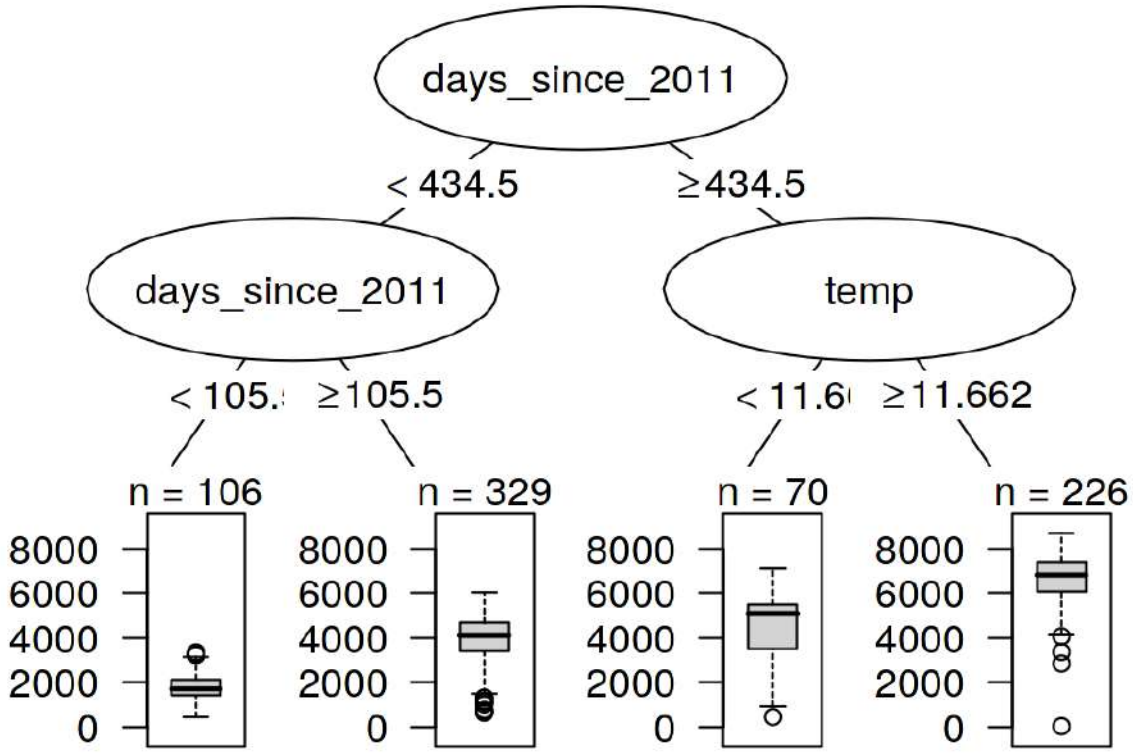
The root node in a decision tree is our starting point. If we were to use the root node to make predictions, it would predict the mean of the outcome of the training data. With the next split, we either subtract or add a term to this sum, depending on the next node in the path. To get to the final prediction, we have to follow the path of the data instance that we want to explain and keep adding to the formula.

$$\hat{f}(x) = \bar{y} + \sum_{d=1}^{D} \text{split.contrib(d,x)} = \bar{y} + \sum_{j=1}^{p} \text{feat.contrib(j,x)}$$

The prediction of an individual instance is the mean of the target outcome plus the sum of all contributions of the D splits that occur between the root node and the terminal node where the instance ends up. We are not interested in the split contributions though, but in the feature contributions. A feature might be used for more than one split or not at all. We can add the contributions for each of the p features and get an interpretation of how much each feature has contributed to a prediction.

Example

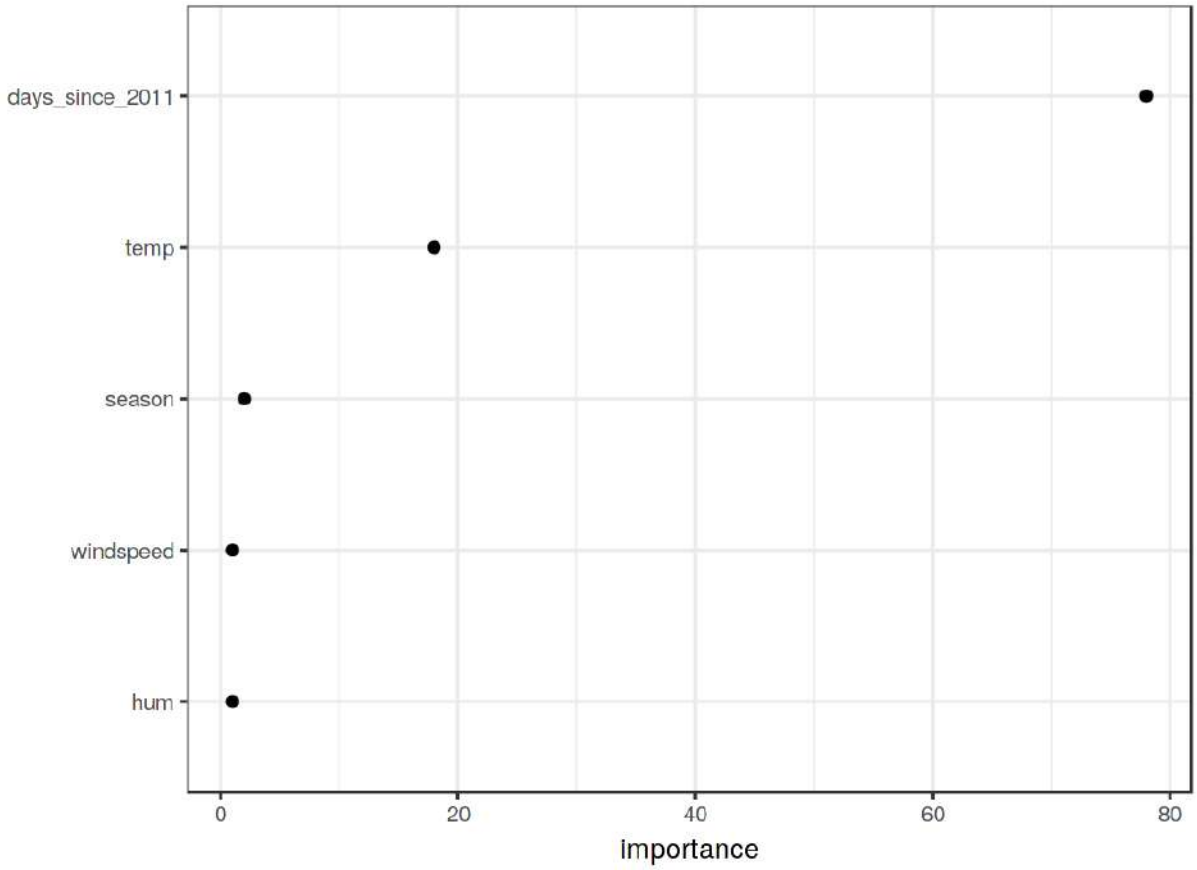
Let us have another look at the bike rental data. We want to predict the number of rented bikes on a certain day with a decision tree. The learned tree looks like this:



Regression tree fitted on the bike rental data. The maximum allowed depth for the tree was set to 2. The trend feature (days since 2011) and the temperature (temp) have been selected for the splits. The boxplots show the distribution of bicycle counts in the terminal node.

The feature importance tells us how much a feature helped to improve the purity of all nodes. Here, the variance was used, since predicting bicycle rentals is a regression task.

The visualized tree shows that both temperature and time trend were used for the splits, but does not quantify which feature was more important. The feature importance measure shows that the time trend is far more important than temperature.



Importance of the features measured by how much the node purity is improved on average.

Advantages

The tree structure is ideal for **capturing interactions** between features in the data.

The data ends up in **distinct groups** that are often easier to understand than points on a multi-dimensional hyperplane as in linear regression. The interpretation is arguably pretty simple.

The tree structure also has a **natural visualization**, with its nodes and edges.

Trees **create good explanations** as defined in the chapter on "Human-Friendly Explanations". The tree structure automatically invites to think about predicted values for individual instances as counterfactuals: "If a feature had been greater / smaller than the split point, the prediction would have been y1 instead of y2. The tree explanations are contrastive, since you can always compare the prediction of an instance with relevant "what if"-scenarios (as defined by the tree) that are simply the other leaf nodes of the tree. If the tree is short, like one to three splits deep, the resulting explanations are selective. A tree with a depth of three requires a maximum of three features and split points to create the explanation for the prediction of an individual instance. The truthfulness of the prediction depends on the predictive performance of the tree. The explanations for short trees

are very simple and general, because for each split the instance falls into either one or the other leaf, and binary decisions are easy to understand.

There is no need to transform features. In linear models, it is sometimes necessary to take the logarithm of a feature. A decision tree works equally well with any monotonic transformation of a feature.

Disadvantages

Trees fail to deal with linear relationships. Any linear relationship between an input feature and the outcome has to be approximated by splits, creating a step function. This is not efficient.

This goes hand in hand with **lack of smoothness**. Slight changes in the input feature can have a big impact on the predicted outcome, which is usually not desirable. Imagine a tree that predicts the value of a house and the tree uses the size of the house as one of the split feature. The split occurs at 100.5 square meters. Imagine user of a house price estimator using your decision tree model: They measure their house, come to the conclusion that the house has 99 square meters, enter it into the price calculator and get a prediction of 200 000 Euro. The users notice that they have forgotten to measure a small storage room with 2 square meters. The storage room has a sloping wall, so they are not sure whether they can count all of the area or only half of it. So they decide to try both 100.0 and 101.0 square meters. The results: The price calculator outputs 200 000 Euro and 205 000 Euro, which is rather unintuitive, because there has been no change from 99 square meters to 100.

Trees are also quite **unstable**. A few changes in the training dataset can create a completely different tree. This is because each split depends on the parent split. And if a different feature is selected as the first split feature, the entire tree structure changes. It does not create confidence in the model if the structure changes so easily.

Decision trees are very interpretable – as long as they are short. **The number of terminal nodes increases quickly with depth**. The more terminal nodes and the deeper the tree, the more difficult it becomes to understand the decision rules of a tree. A depth of 1 means 2 terminal nodes. Depth of 2 means max. 4 nodes. Depth of 3 means max. 8 nodes. The maximum number of terminal nodes in a tree is 2 to the power of the depth.

Software

For the examples in this chapter, I used the `rpart` R package that implements CART (classification and regression trees). CART is implemented in many programming languages, including Python[42]. Arguably, CART is a pretty old and somewhat outdated algorithm and there are some interesting new algorithms for fitting trees. You can find an overview of some R packages for decision trees in the Machine Learning and Statistical Learning CRAN Task View[43] under the keyword "Recursive Partitioning".

[42] https://scikit-learn.org/stable/modules/tree.html
[43] https://cran.r-project.org/web/views/MachineLearning.html

Decision Rules

A decision rule is a simple IF-THEN statement consisting of a condition (also called antecedent) and a prediction. For example: IF it rains today AND if it is April (condition), THEN it will rain tomorrow (prediction). A single decision rule or a combination of several rules can be used to make predictions.

Decision rules follow a general structure: IF the conditions are met THEN make a certain prediction. Decision rules are probably the most interpretable prediction models. Their IF-THEN structure semantically resembles natural language and the way we think, provided that the condition is built from intelligible features, the length of the condition is short (small number of `feature=value` pairs combined with an AND) and there are not too many rules. In programming, it is very natural to write IF-THEN rules. New in machine learning is that the decision rules are learned through an algorithm.

Imagine using an algorithm to learn decision rules for predicting the value of a house (`low`, `medium` or `high`). One decision rule learned by this model could be: If a house is bigger than 100 square meters and has a garden, then its value is high. More formally: IF `size>100 AND garden=1` THEN `value=high`.

Let us break down the decision rule:

- `size>100` is the first condition in the IF-part.
- `garden=1` is the second condition in the IF-part.
- The two conditions are connected with an 'AND' to create a new condition. Both must be true for the rule to apply.
- The predicted outcome (THEN-part) is `value=high`.

A decision rule uses at least one `feature=value` statement in the condition, with no upper limit on how many more can be added with an 'AND'. An exception is the default rule that has no explicit IF-part and that applies when no other rule applies, but more about this later.

The usefulness of a decision rule is usually summarized in two numbers: Support and accuracy.

Support or coverage of a rule: The percentage of instances to which the condition of a rule applies is called the support. Take for example the rule `size=big AND location=good THEN value=high` for predicting house values. Suppose 100 of 1000 houses are big and in a good location, then the support of the rule is 10%. The prediction (THEN-part) is not important for the calculation of support.

Accuracy or confidence of a rule: The accuracy of a rule is a measure of how accurate the rule is in predicting the correct class for the instances to which the condition of the rule applies. For example: Let us say of the 100 houses, where the rule `size=big AND location=good THEN value=high` applies, 85 have `value=high`, 14 have `value=medium` and 1 has `value=low`, then the accuracy of the rule is 85%.

Usually there is a trade-off between accuracy and support: By adding more features to the condition, we can achieve higher accuracy, but lose support.

To create a good classifier for predicting the value of a house you might need to learn not only one rule, but maybe 10 or 20. Then things can get more complicated and you can run into one of the following problems:

- Rules can overlap: What if I want to predict the value of a house and two or more rules apply and they give me contradictory predictions?
- No rule applies: What if I want to predict the value of a house and none of the rules apply?

There are two main strategies for combining multiple rules: Decision lists (ordered) and decision sets (unordered). Both strategies imply different solutions to the problem of overlapping rules.

A **decision list** introduces an order to the decision rules. If the condition of the first rule is true for an instance, we use the prediction of the first rule. If not, we go to the next rule and check if it applies and so on. Decision lists solve the problem of overlapping rules by only returning the prediction of the first rule in the list that applies.

A **decision set** resembles a democracy of the rules, except that some rules might have a higher voting power. In a set, the rules are either mutually exclusive, or there is a strategy for resolving conflicts, such as majority voting, which may be weighted by the individual rule accuracies or other quality measures. Interpretability suffers potentially when several rules apply.

Both decision lists and sets can suffer from the problem that no rule applies to an instance. This can be resolved by introducing a default rule. The default rule is the rule that applies when no other rule applies. The prediction of the default rule is often the most frequent class of the data points which are not covered by other rules. If a set or list of rules covers the entire feature space, we call it exhaustive. By adding a default rule, a set or list automatically becomes exhaustive.

There are many ways to learn rules from data and this book is far from covering them all. This chapter shows you three of them. The algorithms are chosen to cover a wide range of general ideas for learning rules, so all three of them represent very different approaches.

1. **OneR** learns rules from a single feature. OneR is characterized by its simplicity, interpretability and its use as a benchmark.
2. **Sequential covering** is a general procedure that iteratively learns rules and removes the data points that are covered by the new rule. This procedure is used by many rule learning algorithms.
3. **Bayesian Rule Lists** combine pre-mined frequent patterns into a decision list using Bayesian statistics. Using pre-mined patterns is a common approach used by many rule learning algorithms.

Let's start with the simplest approach: Using the single best feature to learn rules.

Learn Rules from a Single Feature (OneR)

The OneR algorithm suggested by Holte (1993)[44] is one of the simplest rule induction algorithms. From all the features, OneR selects the one that carries the most information about the outcome of interest and creates decision rules from this feature.

Despite the name OneR, which stands for "One Rule", the algorithm generates more than one rule: It is actually one rule per unique feature value of the selected best feature. A better name would be OneFeatureRules.

The algorithm is simple and fast:

1. Discretize the continuous features by choosing appropriate intervals.
2. For each feature:
    - Create a cross table between the feature values and the (categorical) outcome.
    - For each value of the feature, create a rule which predicts the most frequent class of the instances that have this particular feature value (can be read from the cross table).
    - Calculate the total error of the rules for the feature.
3. Select the feature with the smallest total error.

OneR always covers all instances of the dataset, since it uses all levels of the selected feature. Missing values can be either treated as an additional feature value or be imputed beforehand.

A OneR model is a decision tree with only one split. The split is not necessarily binary as in CART, but depends on the number of unique feature values.

Let us look at an example how the best feature is chosen by OneR. The following table shows an artificial dataset about houses with information about its value, location, size and whether pets are allowed. We are interested in learning a simple model to predict the value of a house.

location	size	pets	value
good	small	yes	high
good	big	no	high
good	big	no	high
bad	medium	no	medium
good	medium	only cats	medium
good	small	only cats	medium
bad	medium	yes	medium
bad	small	yes	low
bad	medium	yes	low
bad	small	no	low

OneR creates the cross tables between each feature and the outcome:

[44]Holte, Robert C. "Very simple classification rules perform well on most commonly used datasets." Machine learning 11.1 (1993): 63-90.

	value=low	value=medium	value=high
location=bad	3	2	0
location=good	0	2	3

	value=low	value=medium	value=high
size=big	0	0	2
size=medium	1	3	0
size=small	2	1	1

	value=low	value=medium	value=high
pets=no	1	1	2
pets=only cats	0	2	0
pets=yes	2	1	1

For each feature, we go through the table row by row: Each feature value is the IF-part of a rule; the most common class for instances with this feature value is the prediction, the THEN-part of the rule. For example, the size feature with the levels `small`, `medium` and `big` results in three rules. For each feature we calculate the total error rate of the generated rules, which is the sum of the errors. The location feature has the possible values `bad` and `good`. The most frequent value for houses in bad locations is `low` and when we use `low` as a prediction, we make two mistakes, because two houses have a `medium` value. The predicted value of houses in good locations is `high` and again we make two mistakes, because two houses have a `medium` value. The error we make by using the location feature is 4/10, for the size feature it is 3/10 and for the pet feature it is 4/10 . The size feature produces the rules with the lowest error and will be used for the final OneR model:

IF `size=small` THEN `value=small`
IF `size=medium` THEN `value=medium`
IF `size=big` THEN `value=high`

OneR prefers features with many possible levels, because those features can overfit the target more easily. Imagine a dataset that contains only noise and no signal, which means that all features take on random values and have no predictive value for the target. Some features have more levels than others. The features with more levels can now more easily overfit. A feature that has a separate level for each instance from the data would perfectly predict the entire training dataset. A solution would be to split the data into training and validation sets, learn the rules on the training data and evaluate the total error for choosing the feature on the validation set.

Ties are another issue, i.e. when two features result in the same total error. OneR solves ties by either taking the first feature with the lowest error or the one with the lowest p-value of a chi-squared test.

Example

Let us try OneR with real data. We use the cervical cancer classification task to test the OneR algorithm. All continuous input features were discretized into their 5 quantiles. The following rules are created:

Age	prediction
(12.9,27.2]	Healthy
(27.2,41.4]	Healthy
(41.4,55.6]	Healthy
(55.6,69.8]	Healthy
(69.8,84.1]	Healthy

The age feature was chosen by OneR as the best predictive feature. Since cancer is rare, for each rule the majority class and therefore the predicted label is always Healthy, which is rather unhelpful. It does not make sense to use the label prediction in this unbalanced case. The cross table between the 'Age' intervals and Cancer/Healthy together with the percentage of women with cancer is more informative:

	# Cancer	# Healthy	P(Cancer)
Age=(12.9,27.2]	26	477	0.05
Age=(27.2,41.4]	25	290	0.08
Age=(41.4,55.6]	4	31	0.11
Age=(55.6,69.8]	0	1	0.00
Age=(69.8,84.1]	0	4	0.00

But before you start interpreting anything: Since the prediction for every feature and every value is Healthy, the total error rate is the same for all features. The ties in the total error are, by default, resolved by using the first feature from the ones with the lowest error rates (here, all features have 55/858), which happens to be the Age feature.

OneR does not support regression tasks. But we can turn a regression task into a classification task by cutting the continuous outcome into intervals. We use this trick to predict the number of rented bikes with OneR by cutting the number of bikes into its four quartiles (0-25%, 25-50%, 50-75% and 75-100%). The following table shows the selected feature after fitting the OneR model:

mnth	prediction
JAN	[22,3152]
FEB	[22,3152]
MAR	[22,3152]
APR	(3152,4548]
MAY	(5956,8714]
JUN	(4548,5956]
JUL	(5956,8714]
AUG	(5956,8714]
SEP	(5956,8714]
OKT	(5956,8714]
NOV	(3152,4548]
DEZ	[22,3152]

The selected feature is the month. The month feature has (surprise!) 12 feature levels, which is more than most other features have. So there is a danger of overfitting. On the more optimistic side: the month feature can handle the seasonal trend (e.g. less rented bikes in winter) and the predictions

seem sensible.

Now we move from the simple OneR algorithm to a more complex procedure using rules with more complex conditions consisting of several features: Sequential Covering.

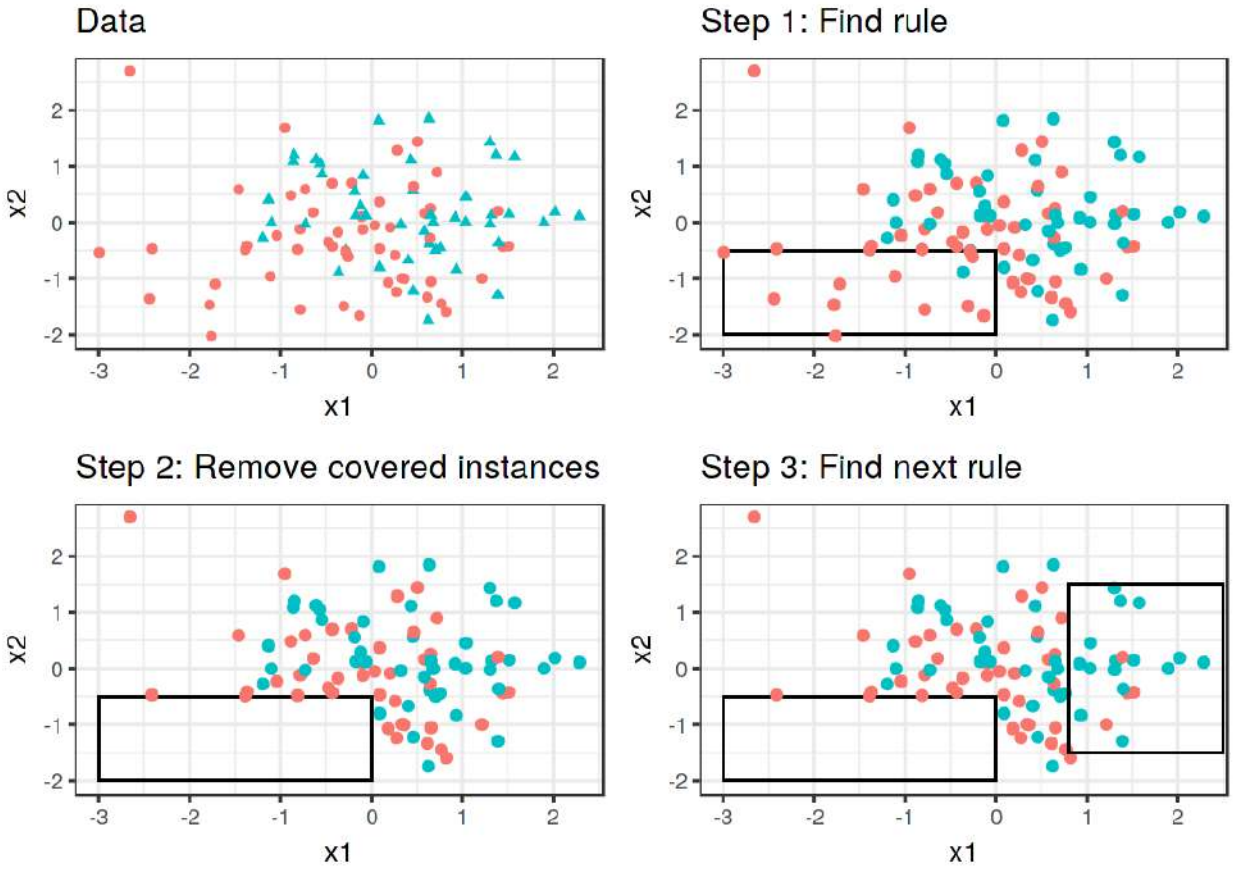
Sequential Covering

Sequential covering is a general procedure that repeatedly learns a single rule to create a decision list (or set) that covers the entire dataset rule by rule. Many rule-learning algorithms are variants of the sequential covering algorithm. This chapter introduces the main recipe and uses RIPPER, a variant of the sequential covering algorithm for the examples.

The idea is simple: First, find a good rule that applies to some of the data points. Remove all data points which are covered by the rule. A data point is covered when the conditions apply, regardless of whether the points are classified correctly or not. Repeat the rule-learning and removal of covered points with the remaining points until no more points are left or another stop condition is met. The result is a decision list. This approach of repeated rule-learning and removal of covered data points is called "separate-and-conquer".

Suppose we already have an algorithm that can create a single rule that covers part of the data. The sequential covering algorithm for two classes (one positive, one negative) works like this:

- Start with an empty list of rules (rlist).
- Learn a rule r.
- While the list of rules is below a certain quality threshold (or positive examples are not yet covered):
  - Add rule r to rlist.
  - Remove all data points covered by rule r.
  - Learn another rule on the remaining data.
- Return the decision list.



The covering algorithm works by sequentially covering the feature space with single rules and removing the data points that are already covered by those rules. For visualization purposes, the features x1 and x2 are continuous, but most rule learning algorithms require categorical features.

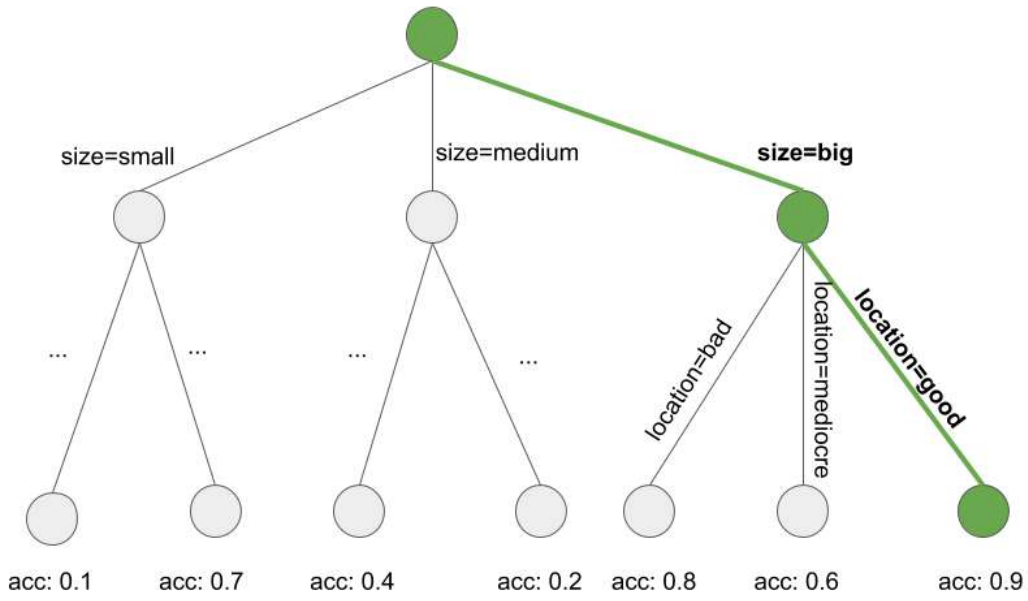
For example: We have a task and dataset for predicting the values of houses from size, location and whether pets are allowed. We learn the first rule, which turns out to be: If `size=big` and `location=good`, then `value=high`. Then we remove all big houses in good locations from the dataset. With the remaining data we learn the next rule. Maybe: If `location=good`, then `value=medium`. Note that this rule is learned on data without big houses in good locations, leaving only medium and small houses in good locations.

For multi-class settings, the approach must be modified. First, the classes are ordered by increasing prevalence. The sequential covering algorithm starts with the least common class, learns a rule for it, removes all covered instances, then moves on to the second least common class and so on. The current class is always treated as the positive class and all classes with a higher prevalence are combined in the negative class. The last class is the default rule. This is also referred to as one-versus-all strategy in classification.

How do we learn a single rule? The OneR algorithm would be useless here, since it would always cover the whole feature space. But there are many other possibilities. One possibility is to learn a single rule from a decision tree with beam search:

- Learn a decision tree (with CART or another tree learning algorithm).
- Start at the root node and recursively select the purest node (e.g. with the lowest misclassification rate).
- The majority class of the terminal node is used as the rule prediction; the path leading to that node is used as the rule condition.

The following figure illustrates the beam search in a tree:



Learning a rule by searching a path through a decision tree. A decision tree is grown to predict the target of interest. We start at the root node, greedily and iteratively follow the path which locally produces the purest subset (e.g. highest accuracy) and add all the split values to the rule condition. We end up with: If `location=good` and `size=big`, then `value=high`.

Learning a single rule is a search problem, where the search space is the space of all possible rules. The goal of the search is to find the best rule according to some criteria. There are many different search strategies: hill-climbing, beam search, exhaustive search, best-first search, ordered search, stochastic search, top-down search, bottom-up search, ...

RIPPER (Repeated Incremental Pruning to Produce Error Reduction) by Cohen (1995)[45] is a variant of the Sequential Covering algorithm. RIPPER is a bit more sophisticated and uses a post-processing phase (rule pruning) to optimize the decision list (or set). RIPPER can run in ordered or unordered mode and generate either a decision list or decision set.

Examples

We will use RIPPER for the examples.

[45]Cohen, William W. "Fast effective rule induction." Machine Learning Proceedings (1995). 115-123.

The RIPPER algorithm does not find any rule in the classification task for cervical cancer.

When we use RIPPER on the regression task to predict bike counts some rules are found. Since RIPPER only works for classification, the bike counts must be turned into a categorical outcome. I achieved this by cutting the bike counts into the quartiles. For example (4548, 5956) is the interval covering predicted bike counts between 4548 and 5956. The following table shows the decision list of learned rules.

rules
(days_since_2011 >= 438) and (temp >= 17) and (temp <= 27) and (hum <= 67) ⇒ cnt=(5956,8714]
(days_since_2011 >= 443) and (temp >= 12) and (weathersit = GOOD) and (hum >= 59) ⇒ cnt=(5956,8714]
(days_since_2011 >= 441) and (windspeed <= 10) and (temp >= 13) ⇒ cnt=(5956,8714]
(temp >= 12) and (hum <= 68) and (days_since_2011 >= 551) ⇒ cnt=(5956,8714]
(days_since_2011 >= 100) and (days_since_2011 <= 434) and (hum <= 72) and (workingday = WORKING DAY) ⇒ cnt=(3152,4548]
(days_since_2011 >= 106) and (days_since_2011 <= 323) ⇒ cnt=(3152,4548]
⇒ cnt=[22,3152]

The interpretation is simple: If the conditions apply, we predict the interval on the right hand side for the number of bikes. The last rule is the default rule that applies when none of the other rules apply to an instance. To predict a new instance, start at the top of the list and check whether a rule applies. When a condition matches, then the right hand side of the rule is the prediction for this instance. The default rule ensures that there is always a prediction.

Bayesian Rule Lists

In this section, I will show you another approach to learning a decision list, which follows this rough recipe:

1. Pre-mine frequent patterns from the data that can be used as conditions for the decision rules.
2. Learn a decision list from a selection of the pre-mined rules.

A specific approach using this recipe is called Bayesian Rule Lists (Letham et. al, 2015)[46] or BRL for short. BRL uses Bayesian statistics to learn decision lists from frequent patterns which are pre-mined with the FP-tree algorithm (Borgelt 2005)[47]

But let us start slowly with the first step of BRL.

Pre-mining of frequent patterns

A frequent pattern is the frequent (co-)occurrence of feature values. As a pre-processing step for the BRL algorithm, we use the features (we do not need the target outcome in this step) and extract

[46]Letham, Benjamin, et al. "Interpretable classifiers using rules and Bayesian analysis: Building a better stroke prediction model." The Annals of Applied Statistics 9.3 (2015): 1350-1371.

[47]Borgelt, C. "An implementation of the FP-growth algorithm." Proceedings of the 1st International Workshop on Open Source Data Mining Frequent Pattern Mining Implementations - OSDM '05, 1–5. http://doi.org/10.1145/1133905.1133907 (2005).

frequently occurring patterns from them. A pattern can be a single feature value such as `size=medium` or a combination of feature values such as `size=medium AND location=bad`.

The frequency of a pattern is measured with its support in the dataset:

$$Support(x_j = A) = \frac{1}{n}\sum_{i=1}^{n} I(x_j^{(i)} = A)$$

where A is the feature value, n the number of data points in the dataset and I the indicator function that returns 1 if the feature x_j of the instance i has level A otherwise 0. In a dataset of house values, if 20% of houses have no balcony and 80% have one or more, then the support for the pattern `balcony=0` is 20%. Support can also be measured for combinations of feature values, for example for `balcony=0 AND pets=allowed`.

There are many algorithms to find such frequent patterns, for example Apriori or FP-Growth. Which you use does not matter much, only the speed at which the patterns are found is different, but the resulting patterns are always the same.

I will give you a rough idea of how the Apriori algorithm works to find frequent patterns. Actually the Apriori algorithm consists of two parts, where the first part finds frequent patterns and the second part builds association rules from them. For the BRL algorithm, we are only interested in the frequent patterns that are generated in the first part of Apriori.

In the first step, the Apriori algorithm starts with all feature values that have a support greater than the minimum support defined by the user. If the user says that the minimum support should be 10% and only 5% of the houses have `size=big`, we would remove that feature value and keep only `size=medium` and `size=small` as patterns. This does not mean that the houses are removed from the data, it just means that `size=big` is not returned as frequent pattern. Based on frequent patterns with a single feature value, the Apriori algorithm iteratively tries to find combinations of feature values of increasingly higher order. Patterns are constructed by combining `feature=value` statements with a logical AND, e.g. `size=medium AND location=bad`. Generated patterns with a support below the minimum support are removed. In the end we have all the frequent patterns. Any subset of a frequent pattern is frequent again, which is called the Apriori property. It makes sense intuitively: By removing a condition from a pattern, the reduced pattern can only cover more or the same number of data points, but not less. For example, if 20% of the houses are `size=medium and location=good`, then the support of houses that are only `size=medium` is 20% or greater. The Apriori property is used to reduce the number of patterns to be inspected. Only in the case of frequent patterns we have to check patterns of higher order.

Now we are done with pre-mining conditions for the Bayesian Rule List algorithm. But before we move on to the second step of BRL, I would like to hint at another way for rule-learning based on pre-mined patterns. Other approaches suggest including the outcome of interest into the frequent pattern mining process and also executing the second part of the Apriori algorithm that builds IF-THEN rules. Since the algorithm is unsupervised, the THEN-part also contains feature values we are not interested in. But we can filter by rules that have only the outcome of interest in the THEN-part.

These rules already form a decision set, but it would also be possible to arrange, prune, delete or recombine the rules.

In the BRL approach however, we work with the frequent patterns and learn the THEN-part and how to arrange the patterns into a decision list using Bayesian statistics.

Learning Bayesian Rule Lists

The goal of the BRL algorithm is to learn an accurate decision list using a selection of the pre-mined conditions, while prioritizing lists with few rules and short conditions. BRL addresses this goal by defining a distribution of decision lists with prior distributions for the length of conditions (preferably shorter rules) and the number of rules (preferably a shorter list).

The posteriori probability distribution of lists makes it possible to say how likely a decision list is, given assumptions of shortness and how well the list fits the data. Our goal is to find the list that maximizes this posterior probability. Since it is not possible to find the exact best list directly from the distributions of lists, BRL suggests the following recipe:
1) Generate an initial decision list, which is randomly drawn from the priori distribution.
2) Iteratively modify the list by adding, switching or removing rules, ensuring that the resulting lists follow the posterior distribution of lists.
3) Select the decision list from the sampled lists with the highest probability according to the posteriori distribution.

Let us go over the algorithm more closely: The algorithm starts with pre-mining feature value patterns with the FP-Growth algorithm. BRL makes a number of assumptions about the distribution of the target and the distribution of the parameters that define the distribution of the target. (That's Bayesian statistic.) If you are unfamiliar with Bayesian statistics, do not get too caught up in the following explanations. It is important to know that the Bayesian approach is a way to combine existing knowledge or requirements (so-called priori distributions) while also fitting to the data. In the case of decision lists, the Bayesian approach makes sense, since the prior assumptions nudges the decision lists to be short with short rules.

The goal is to sample decision lists d from the posteriori distribution:

$$\underbrace{p(d|x,y,A,\alpha,\lambda,\eta)}_{posteriori} \propto \underbrace{p(y|x,d,\alpha)}_{likelihood} \cdot \underbrace{p(d|A,\lambda,\eta)}_{priori}$$

where d is a decision list, x are the features, y is the target, A the set of pre-mined conditions, λ the prior expected length of the decision lists, η the prior expected number of conditions in a rule, α the prior pseudo-count for the positive and negative classes which is best fixed at (1,1).

$$p(d|x,y,A,\alpha,\lambda,\eta)$$

quantifies how probable a decision list is, given the observed data and the priori assumptions. This is proportional to the likelihood of the outcome y given the decision list and the data times the probability of the list given prior assumptions and the pre-mined conditions.

$$p(y|x, d, \alpha)$$

is the likelihood of the observed y, given the decision list and the data. BRL assumes that y is generated by a Dirichlet-Multinomial distribution. The better the decision list d explains the data, the higher the likelihood.

$$p(d|A, \lambda, \eta)$$

is the prior distribution of the decision lists. It multiplicatively combines a truncated Poisson distribution (parameter λ) for the number of rules in the list and a truncated Poisson distribution (parameter η) for the number of feature values in the conditions of the rules.

A decision list has a high posterior probability if it explains the outcome y well and is also likely according to the prior assumptions.

Estimations in Bayesian statistics are always a bit tricky, because we usually cannot directly calculate the correct answer, but we have to draw candidates, evaluate them and update our posteriori estimates using the Markov chain Monte Carlo method. For decision lists, this is even more tricky, because we have to draw from the distribution of decision lists. The BRL authors propose to first draw an initial decision list and then iteratively modify it to generate samples of decision lists from the posterior distribution of the lists (a Markov chain of decision lists). The results are potentially dependent on the initial decision list, so it is advisable to repeat this procedure to ensure a great variety of lists. The default in the software implementation is 10 times. The following recipe tells us how to draw an initial decision list:

- Pre-mine patterns with FP-Growth.
- Sample the list length parameter m from a truncated Poisson distribution.
- For the default rule: Sample the Dirichlet-Multinomial distribution parameter θ_0 of the target value (i.e. the rule that applies when nothing else applies).
- For decision list rule j=1,...,m, do:
  - Sample the rule length parameter l (number of conditions) for rule j.
  - Sample a condition of length l_j from the pre-mined conditions.
  - Sample the Dirichlet-Multinomial distribution parameter for the THEN-part (i.e. for the distribution of the target outcome given the rule)
- For each observation in the dataset:
  - Find the rule from the decision list that applies first (top to bottom).
  - Draw the predicted outcome from the probability distribution (Binomial) suggested by the rule that applies.

The next step is to generate many new lists starting from this initial sample to obtain many samples from the posterior distribution of decision lists.

The new decision lists are sampled by starting from the initial list and then randomly either moving a rule to a different position in the list or adding a rule to the current decision list from the pre-mined conditions or removing a rule from the decision list. Which of the rules is switched, added or deleted is chosen at random. At each step, the algorithm evaluates the posteriori probability of the decision list (mixture of accuracy and shortness). The Metropolis Hastings algorithm ensures that we sample decision lists that have a high posterior probability. This procedure provides us with many samples from the distribution of decision lists. The BRL algorithm selects the decision list of the samples with the highest posterior probability.

Examples

That is it with the theory, now let's see the BRL method in action. The examples use a faster variant of BRL called Scalable Bayesian Rule Lists (SBRL) by Yang et. al (2017) [48]. We use the SBRL algorithm to predict the risk for cervical cancer. I first had to discretize all input features for the SBRL algorithm to work. For this purpose I binned the continuous features based on the frequency of the values by quantiles.

We get the following rules:

rules
If {STDs=1} (rule[259]) then positive probability = 0.16049383
else if {Hormonal.Contraceptives..years.=[0,10)} (rule[82]) then positive probability = 0.04685408
else (default rule) then positive probability = 0.27777778

Note that we get sensible rules, since the prediction on the THEN-part is not the class outcome, but the predicted probability for cancer.

The conditions were selected from patterns that were pre-mined with the FP-Growth algorithm. The following table displays the pool of conditions the SBRL algorithm could choose from for building a decision list. The maximum number of feature values in a condition I allowed as a user was two. Here is a sample of ten patterns:

pre-mined conditions
First.sexual.intercourse=[17.3,24.7),STDs=1
Hormonal.Contraceptives=0,STDs=0
Num.of.pregnancies=[0,3.67),STDs..Number.of.diagnosis=[0,1)
Smokes=1
First.sexual.intercourse=[10,17.3)
Smokes=1,STDs..Number.of.diagnosis=[0,1)
STDs..number.=[1.33,2.67)
Num.of.pregnancies=[3.67,7.33)
Num.of.pregnancies=[3.67,7.33),IUD..years.=[0,6.33)
Age=[13,36.7),STDs..Number.of.diagnosis=[1,2)

Next, we apply the SBRL algorithm to the bike rental prediction task. This only works if the

[48] Yang, Hongyu, Cynthia Rudin, and Margo Seltzer. "Scalable Bayesian rule lists." Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

regression problem of predicting bike counts is converted into a binary classification task. I have arbitrarily created a classification task by creating a label that is 1 if the number of bikes exceeds 4000 bikes on a day, else 0.

The following list was learned by SBRL:

rules
If {yr=2011,temp=[-5.22,7.35)} (rule[718]) then positive probability = 0.01041667
else if {yr=2012,temp=[7.35,19.9)} (rule[823]) then positive probability = 0.88125000
else if {yr=2012,temp=[19.9,32.5]} (rule[816]) then positive probability = 0.99253731
else if {season=SPRING} (rule[351]) then positive probability = 0.06410256
else if {yr=2011,temp=[7.35,19.9)} (rule[730]) then positive probability = 0.44444444
else (default rule) then positive probability = 0.79746835

Let us predict the probability that the number of bikes will exceed 4000 for a day in 2012 with a temperature of 17 degrees Celsius. The first rule does not apply, since it only applies for days in 2011. The second rule applies, because the day is in 2012 and 17 degrees lies in the interval `[7.35,19.9)`. Our prediction for the probability is that more than 4000 bikes are rented is 88%.

Advantages

This section discusses the benefits of IF-THEN rules in general.

IF-THEN rules are **easy to interpret**. They are probably the most interpretable of the interpretable models. This statement only applies if the number of rules is small, the conditions of the rules are short (maximum 3 I would say) and if the rules are organized in a decision list or a non-overlapping decision set.

Decision rules can be **as expressive as decision trees**, **while being more compact**. Decision trees often also suffer from replicated sub-trees, that is, when the splits in a left and a right child node have the same structure.

The **prediction with IF-THEN rules is fast**, since only a few binary statements need to be checked to determine which rules apply.

Decision rules are **robust** against monotonous transformations of the input features, because only the threshold in the conditions changes. They are also robust against outliers, since it only matters if a condition applies or not.

IF-THEN rules usually generate sparse models, which means that not many features are included. They **select only the relevant features** for the model. For example, a linear model assigns a weight to every input feature by default. Features that are irrelevant can simply be ignored by IF-THEN rules.

Simple rules like from OneR **can be used as baseline** for more complex algorithms.

Disadvantages

This section deals with the disadvantages of IF-THEN rules in general.

The research and literature for IF-THEN rules focuses on classification and almost **completely neglects regression**. While you can always divide a continuous target into intervals and turn it into a classification problem, you always lose information. In general, approaches are more attractive if they can be used for both regression and classification.

Often the **features also have to be categorical**. That means numeric features must be categorized if you want to use them. There are many ways to cut a continuous feature into intervals, but this is not trivial and comes with many questions without clear answers. How many intervals should the feature be divided into? What is the splitting criteria: Fixed interval lengths, quantiles or something else? Categorizing continuous features is a non-trivial issue that is often neglected and people just use the next best method (like I did in the examples).

Many of the older rule-learning algorithms are prone to overfitting. The algorithms presented here all have at least some safeguards to prevent overfitting: OneR is limited because it can only use one feature (only problematic if the feature has too many levels or if there are many features, which equates to the multiple testing problem), RIPPER does pruning and Bayesian Rule Lists impose a prior distribution on the decision lists.

Decision rules are **bad in describing linear relationships** between features and output. That is a problem they share with the decision trees. Decision trees and rules can only produce step-like prediction functions, where changes in the prediction are always discrete steps and never smooth curves. This is related to the issue that the inputs have to be categorical. In decision trees, they are implicitly categorized by splitting them.

Software and Alternatives

OneR is implemented in the R package OneR[49], which was used for the examples in this book. OneR is also implemented in the Weka machine learning library[50] and as such available in Java, R and Python. RIPPER is also implemented in Weka. For the examples, I used the R implementation of JRIP in the RWeka package[51]. SBRL is available as R package[52] (which I used for the examples), in Python[53] or as C implementation[54].

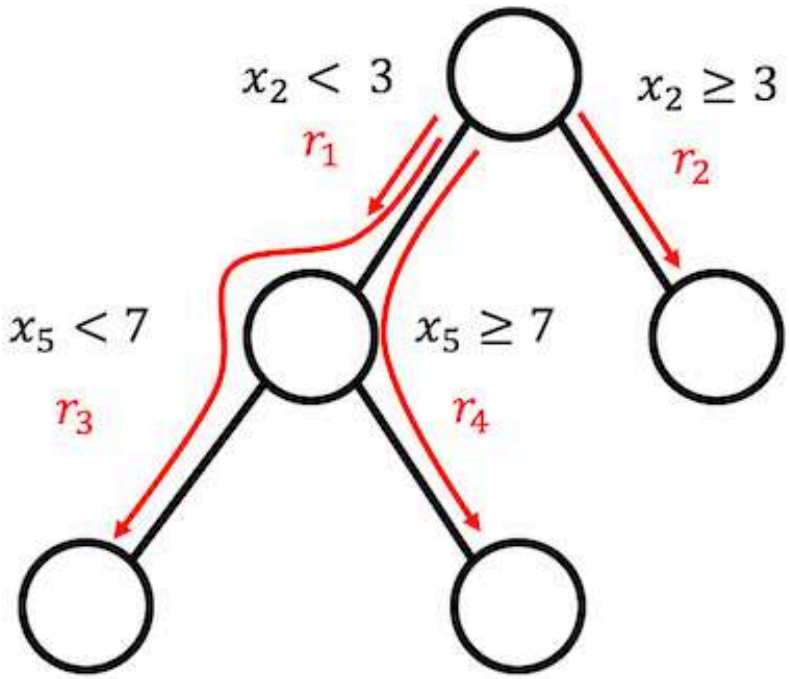
I will not even try to list all alternatives for learning decision rule sets and lists, but will point to some summarizing work. I recommend the book "Foundations of Rule Learning" by Fuernkranz et. al (2012)[55]. It is an extensive work on learning rules, for those who want to delve deeper into the topic. It provides a holistic framework for thinking about learning rules and presents many rule learning algorithms. I also recommend to checkout the Weka rule learners[56], which implement RIPPER, M5Rules, OneR, PART and many more. IF-THEN rules can be used in linear models as described in this book in the chapter about the RuleFit algorithm.

[49] https://cran.r-project.org/web/packages/OneR/
[50] (https://www.eecs.yorku.ca/tdb/_doc.php/userg/sw/weka/doc/weka/classifiers/rules/package-summary.html)
[51] https://cran.r-project.org/web/packages/RWeka/index.html
[52] https://cran.r-project.org/web/packages/sbrl/index.html
[53] https://github.com/datascienceinc/Skater
[54] https://github.com/Hongyuy/sbrlmod
[55] Fürnkranz, Johannes, Dragan Gamberger, and Nada Lavrač. "Foundations of rule learning." Springer Science & Business Media, (2012).
[56] http://weka.sourceforge.net/doc.dev/weka/classifiers/rules/package-summary.html

RuleFit

The RuleFit algorithm by Friedman and Popescu (2008)[57] learns sparse linear models that include automatically detected interaction effects in the form of decision rules.

The linear regression model does not account for interactions between features. Would it not be convenient to have a model that is as simple and interpretable as linear models, but also integrates feature interactions? RuleFit fills this gap. RuleFit learns a sparse linear model with the original features and also a number of new features that are decision rules. These new features capture interactions between the original features. RuleFit automatically generates these features from decision trees. Each path through a tree can be transformed into a decision rule by combining the split decisions into a rule. The node predictions are discarded and only the splits are used in the decision rules:



4 rules can be generated from a tree with 3 terminal nodes.

Where do those decision trees come from? The trees are trained to predict the outcome of interest. This ensures that the splits are meaningful for the prediction task. Any algorithm that generates a lot of trees can be used for RuleFit, for example a random forest. Each tree is decomposed into decision rules that are used as additional features in a sparse linear regression model (Lasso).

The RuleFit paper uses the Boston housing data to illustrate this: The goal is to predict the median house value of a Boston neighborhood. One of the rules generated by RuleFit is: IF `number of rooms`

[57]Friedman, Jerome H, and Bogdan E Popescu. "Predictive learning via rule ensembles." The Annals of Applied Statistics. JSTOR, 916â€“54. (2008).

> 6.64 AND concentration of nitric oxide <0.67 THEN 1 ELSE 0.

RuleFit also comes with a feature importance measure that helps to identify linear terms and rules that are important for the predictions. Feature importance is calculated from the weights of the regression model. The importance measure can be aggregated for the original features (which are used in their "raw" form and possibly in many decision rules).

RuleFit also introduces partial dependence plots to show the average change in prediction by changing a feature. The partial dependence plot is a model-agnostic method that can be used with any model, and is explained in the book chapter on partial dependence plots.

Interpretation and Example

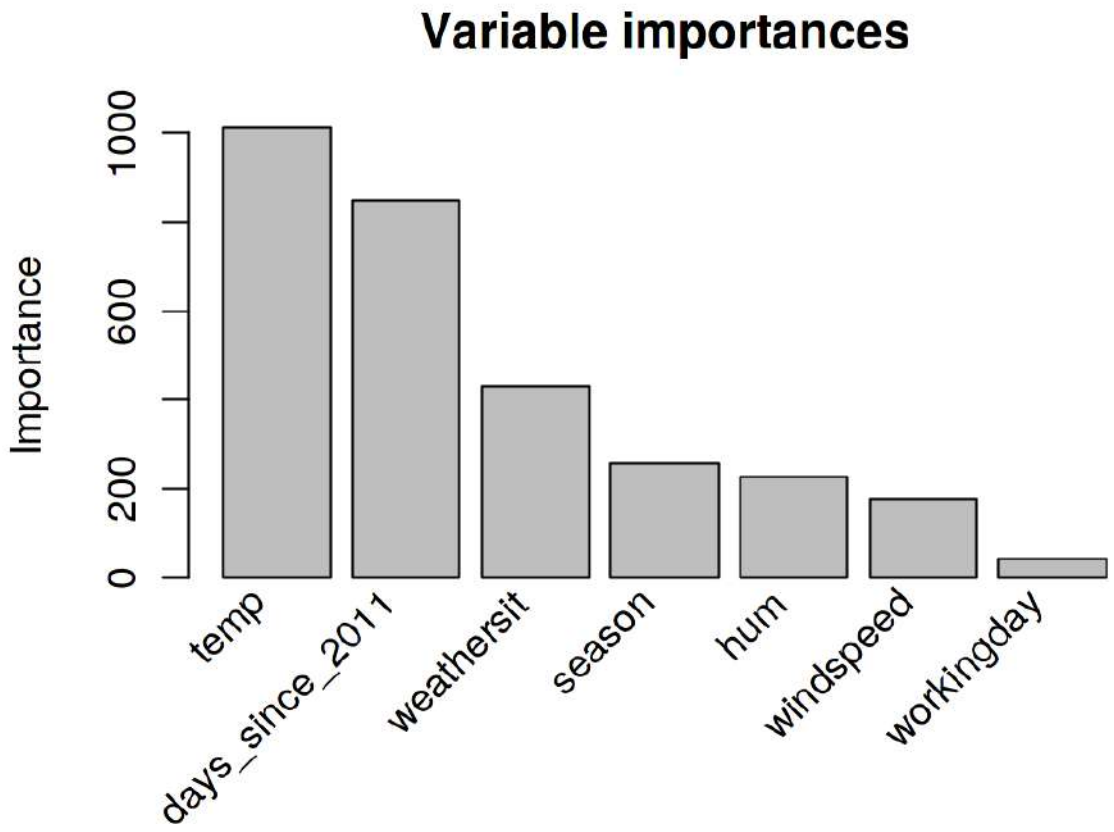
Since RuleFit estimates a linear model in the end, the interpretation is the same as for "normal" linear models. The only difference is that the model has new features derived from decision rules. Decision rules are binary features: A value of 1 means that all conditions of the rule are met, otherwise the value is 0. For linear terms in RuleFit, the interpretation is the same as in linear regression models: If the feature increases by one unit, the predicted outcome changes by the corresponding feature weight.

In this example, we use RuleFit to predict the number of rented bicycles on a given day. The table shows five of the rules that were generated by RuleFit, along with their Lasso weights and importances. The calculation is explained later in the chapter.

Description	Weight	Importance
days_since_2011 > 111 & weathersit in ("GOOD", "MISTY")	664	253
37.25 <= hum <= 90	-17	227
days_since_2011 > 428 & temp > 5	460	225
temp > 13 & days_since_2011 > 554	550	194
temp > 8 & weathersit in ("GOOD", "MISTY")	409	188

The most important rule was: "days_since_2011 > 111 & weathersit in ("GOOD", "MISTY")" and the corresponding weight is 664. The interpretation is: If days_since_2011 > 111 & weathersit in ("GOOD", "MISTY"), then the predicted number of bikes increases by 664, when all other feature values remain fixed. In total, 278 such rules were created from the original 8 features. Quite a lot! But thanks to Lasso, only 39 of the 278 have a weight different from 0.

Computing the global feature importances reveals that temperature and time trend are the most important features:



Feature importance measures for a RuleFit model predicting bike counts. The most important features for the predictions were temperature and time trend.

The feature importance measurement includes the importance of the raw feature term and all the decision rules in which the feature appears.

Interpretation template

The interpretation is analogous to linear models: The predicted outcome changes by β_j if feature x_j changes by one unit, provided all other features remain unchanged. The weight interpretation of a decision rule is a special case: If all conditions of a decision rule r_k apply, the predicted outcome changes by α_k (the learned weight of rule r_k in the linear model).

For classification (using logistic regression instead of linear regression): If all conditions of the decision rule r_k apply, the odds for event vs. no-event changes by a factor of α_k.

Theory

Let us dive deeper into the technical details of the RuleFit algorithm. RuleFit consists of two components: The first component creates "rules" from decision trees and the second component fits a linear model with the original features and the new rules as input (hence the name "RuleFit").

Step 1: Rule generation

What does a rule look like? The rules generated by the algorithm have a simple form. For example: IF `x2 < 3` AND `x5 < 7` THEN 1 ELSE 0. The rules are constructed by decomposing decision trees: Any path to a node in a tree can be converted to a decision rule. The trees used for the rules are fitted to predict the target outcome. Therefore the splits and resulting rules are optimized to predict the outcome you are interested in. You simply chain the binary decisions that lead to a certain node with "AND", and voilÃ , you have a rule. It is desirable to generate a lot of diverse and meaningful rules. Gradient boosting is used to fit an ensemble of decision trees by regressing or classifying y with your original features X. Each resulting tree is converted into multiple rules. Not only boosted trees, but any tree ensemble algorithm can be used to generate the trees for RuleFit. A tree ensemble can be described with this general formula:

$$f(x) = a_0 + \sum_{m=1}^{M} a_m f_m(X)$$

M is the number of trees and $f_m(x)$ is the prediction function of the m-th tree. The α's are the weights. Bagged ensembles, random forest, AdaBoost and MART produce tree ensembles and can be used for RuleFit.

We create the rules from all trees of the ensemble. Each rule r_m takes the form of:

$$r_m(x) = \prod_{j \in \mathrm{T}_m} I(x_j \in s_{jm})$$

where T_m is the set of features used in the m-th tree, I is the indicator function that is 1 when feature x_j is in the specified subset of values s for the j-th feature (as specified by the tree splits) and 0 otherwise. For numerical features, s_{jm} is an interval in the value range of the feature. The interval looks like one of the two cases:

$$x_{s_{jm},\text{lower}} < x_j$$

$$x_j < x_{s_{jm},upper}$$

Further splits in that feature possibly lead to more complicated intervals. For categorical features the subset s contains some specific categories of the feature.

A made up example for the bike rental dataset:

$$r_{17}(x) = I(x_{\text{temp}} < 15) \cdot I(x_{\text{weather}} \in \{\text{good}, \text{cloudy}\}) \cdot I(10 \leq x_{\text{windspeed}} < 20)$$

This rule returns 1 if all three conditions are met, otherwise 0. RuleFit extracts all possible rules from a tree, not only from the leaf nodes. So another rule that would be created is:

$$r_{18}(x) = I(x_{\text{temp}} < 15) \cdot I(x_{\text{weather}} \in \{\text{good}, \text{cloudy}\}$$

Altogether, the number of rules created from an ensemble of M trees with t_m terminal nodes each is:

$$K = \sum_{m=1}^{M} 2(t_m - 1)$$

A trick introduced by the RuleFit authors is to learn trees with random depth so that many diverse rules with different lengths are generated. Note that we discard the predicted value in each node and only keep the conditions that lead us to a node and then we create a rule from it. The weighting of the decision rules is done in step 2 of RuleFit.

Another way to see step 1: RuleFit generates a new set of features from your original features. These features are binary and can represent quite complex interactions of your original features. The rules are chosen to maximize the prediction task. The rules are automatically generated from the covariates matrix X. You can simply see the rules as new features based on your original features.

Step 2: Sparse linear model

You get MANY rules in step 1. Since the first step can be seen as only a feature transformation, you are still not done with fitting a model. Also, you want to reduce the number of rules. In addition to the rules, all your "raw" features from your original dataset will also be used in the sparse linear model. Every rule and every original feature becomes a feature in the linear model and gets a weight estimate. The original raw features are added because trees fail at representing simple linear relationships between y and x. Before we train a sparse linear model, we winsorize the original features so that they are more robust against outliers:

$$l_j^*(x_j) = min(\delta_j^+, max(\delta_j^-, x_j))$$

where δ_j^- and δ_j^+ are the δ quantiles of the data distribution of feature x_j. A choice of 0.05 for δ means that any value of feature x_j that is in the 5% lowest or 5% highest values will be set to the quantiles at 5% or 95% respectively. As a rule of thumb, you can choose δ = 0.025. In addition, the linear terms have to be normalized so that they have the same prior importance as a typical decision rule:

$$l_j(x_j) = 0.4 \cdot l_j^*(x_j)/std(l_j^*(x_j))$$

The 0.4 is the average standard deviation of rules with a uniform support distribution of $s_k \sim U(0, 1)$.

We combine both types of features to generate a new feature matrix and train a sparse linear model with Lasso, with the following structure:

$$\hat{f}(x) = \hat{\beta}_0 + \sum_{k=1}^{K} \hat{\alpha}_k r_k(x) + \sum_{j=1}^{p} \hat{\beta}_j l_j(x_j)$$

where $\hat{\alpha}$ is the estimated weight vector for the rule features and $\hat{\beta}$ the weight vector for the original features. Since RuleFit uses Lasso, the loss function gets the additional constraint that forces some of the weights to get a zero estimate:

$$(\{\hat{\alpha}\}_1^K, \{\hat{\beta}\}_0^p) = argmin_{\{\hat{\alpha}\}_1^K, \{\hat{\beta}\}_0^p} \sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)})) + \lambda \cdot \left(\sum_{k=1}^{K} |\alpha_k| + \sum_{j=1}^{p} |b_j| \right)$$

The result is a linear model that has linear effects for all of the original features and for the rules. The interpretation is the same as for linear models, the only difference is that some features are now binary rules.

Step 3 (optional): Feature importance

For the linear terms of the original features, the feature importance is measured with the standardized predictor:

$$I_j = |\hat{\beta}_j| \cdot std(l_j(x_j))$$

where β_j is the weight from the Lasso model and $std(l_j(x_j))$ is the standard deviation of the linear term over the data.

For the decision rule terms, the importance is calculated with the following formula:

$$I_k = |\hat{\alpha}_k| \cdot \sqrt{s_k(1 - s_k)}$$

where $\hat{\alpha}_k$ is the associated Lasso weight of the decision rule and s_k is the support of the feature in the data, which is the percentage of data points to which the decision rule applies (where $r_k(x) = 0$):

$$s_k = \frac{1}{n} \sum_{i=1}^{n} r_k(x^{(i)})$$

A feature occurs as a linear term and possibly also within many decision rules. How do we measure the total importance of a feature? The importance $J_j(x)$ of a feature can be measured for each individual prediction:

$$J_j(x) = I_l(x) + \sum_{x_j \in r_k} I_k(x)/m_k$$

where I_l is the importance of the linear term and I_k the importance of the decision rules in which x_j

appears, and m_k is the number of features constituting the rule r_k. Adding the feature importance from all instances gives us the global feature importance:

$$J_j(X) = \sum_{i=1}^{n} J_j(x^{(i)})$$

It is possible to select a subset of instances and calculate the feature importance for this group.

Advantages

RuleFit automatically adds **feature interactions** to linear models. Therefore, it solves the problem of linear models that you have to add interaction terms manually and it helps a bit with the issue of modeling nonlinear relationships.

RuleFit can handle both classification and regression tasks.

The rules created are easy to interpret, because they are binary decision rules. Either the rule applies to an instance or not. Good interpretability is only guaranteed if the number of conditions within a rule is not too large. A rule with 1 to 3 conditions seems reasonable to me. This means a maximum depth of 3 for the trees in the tree ensemble.

Even if there are many rules in the model, they do not apply to every instance. For an individual instance only a handful of rules apply (= have a non-zero weights). This improves local interpretability.

RuleFit proposes a bunch of useful diagnostic tools. These tools are model-agnostic, so you can find them in the model-agnostic section of the book: feature importance, partial dependence plots and feature interactions.

Disadvantages

Sometimes RuleFit creates many rules that get a non-zero weight in the Lasso model. The interpretability degrades with increasing number of features in the model. A promising solution is to force feature effects to be monotonic, meaning that an increase of a feature has to lead to an increase of the prediction.

An anecdotal drawback: The papers claim a good performance of RuleFit – often close to the predictive performance of random forests! – but in the few cases where I tried it personally, the performance was disappointing. Just try it out for your problem and see how it performs.

The end product of the RuleFit procedure is a linear model with additional fancy features (the decision rules). But since it is a linear model, the weight interpretation is still unintuitive. It comes with the same "footnote" as a usual linear regression model: "... given all features are fixed." It gets a bit more tricky when you have overlapping rules. For example, one decision rule (feature) for the bicycle prediction could be: "temp > 10" and another rule could be "temp > 15 & weather='GOOD'". If the weather is good and the temperature is above 15 degrees, the temperature is automatically greater

then 10. In the cases where the second rule applies, the first rule applies as well. The interpretation of the estimated weight for the second rule is: "Assuming all other features remain fixed, the predicted number of bikes increases by β_2 when the weather is good and temperature above 15 degrees.". But, now it becomes really clear that the 'all other feature fixed' is problematic, because if rule 2 applies, also rule 1 applies and the interpretation is nonsensical.

Software and Alternative

The RuleFit algorithm is implemented in R by Fokkema and Christoffersen (2017)[58] and you can find a Python version on Github[59].

A very similar framework is scope-rules[60], a Python module that also extracts rules from ensembles. It differs in the way it learns the final rules: First, similar and duplicate rules are removed. Then scope-rules chooses rules based on recall and precision instead of relying on Lasso.

[58]Fokkema, Marjolein, and Benjamin Christoffersen. "Pre: Prediction rule ensembles". https://CRAN.R-project.org/package=pre (2017).
[59]https://github.com/christophM/rulefit
[60]https://github.com/scikit-learn-contrib/skope-rules

Other Interpretable Models

The list of interpretable models is constantly growing and of unknown size. It includes simple models such as linear models, decision trees and naive Bayes, but also more complex ones that combine or modify non-interpretable machine learning models to make them more interpretable. Especially publications on the latter type of models are currently being produced at high frequency and it is hard to keep up with developments. The book teases only the Naive Bayes classifier and k-nearest neighbors in this chapter.

Naive Bayes Classifier

The Naive Bayes classifier uses the Bayes' theorem of conditional probabilities. For each feature, it calculates the probability for a class depending on the value of the feature. The Naive Bayes classifier calculates the class probabilities for each feature independently, which is equivalent to a strong (= naive) assumption of independence of the features. Naive Bayes is a conditional probability model and models the probability of a class C_k as follows:

$$P(C_k|x) = \frac{1}{Z}P(C_k)\prod_{i=1}^{n}P(x_i|C_k)$$

The term Z is a scaling parameter that ensures that the sum of probabilities for all classes is 1 (otherwise they would not be probabilities). The conditional probability of a class is the class probability times the probability of each feature given the class, normalized by Z. This formula can be derived by using the Bayes' theorem.

Naive Bayes is an interpretable model because of the independence assumption. It can be interpreted on the modular level. It is very clear for each feature how much it contributes towards a certain class prediction, since we can interpret the conditional probability.

K-Nearest Neighbors

The k-nearest neighbor method can be used for regression and classification and uses the nearest neighbors of a data point for prediction. For classification, the k-nearest neighbor method assigns the most common class of the nearest neighbors of an instance. For regression, it takes the average of the outcome of the neighbors. The tricky parts are finding the right k and deciding how to measure the distance between instances, which ultimately defines the neighborhood.

The k-nearest neighbor model differs from the other interpretable models presented in this book because it is an instance-based learning algorithm. How can k-nearest neighbors be interpreted? First of all, there are no parameters to learn, so there is no interpretability on a modular level. Furthermore, there is a lack of global model interpretability because the model is inherently local and there are no global weights or structures explicitly learned. Maybe it is interpretable at the

local level? To explain a prediction, you can always retrieve the k neighbors that were used for the prediction. Whether the model is interpretable depends solely on the question whether you can 'interpret' a single instance in the dataset. If an instance consists of hundreds or thousands of features, then it is not interpretable, I would argue. But if you have few features or a way to reduce your instance to the most important features, presenting the k-nearest neighbors can give you good explanations.