

```
#hide
! [-e /content] && pip install -Uqq fastbook kaggle waterfallcharts treeinterpreter dtreeviz
import fastbook
fastbook.setup_book()

#hide
from fastbook import *
from pandas.api.types import is_string_dtype, is_numeric_dtype, is_categorical_dtype
from fastai.tabular.all import *
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor from dtreeviz.trees import *
from IPython.display import Image, display_svg, SVG
pd.options.display.max_rows = 20
pd.options.display.max_columns = 8
```

In []:

Tabular Modeling Deep Dive

タブラーーモデリングのディープダイブ

Tabular modeling takes data in the form of a table (like a spreadsheet or CSV). The objective is to predict the value in one column based on the values in the other columns. In this chapter we will not only look at deep learning but also more general machine learning techniques like random forests, as they can give better results depending on your problem.

表形式モデリングは、（スプレッドシートや CSV のような）表の形でデータを受け取ります。目的は、他の列の値に基づいて、ある列の値を予測することです。この章では、ディープラーニングだけでなく、ランダムフォレストのような一般的な機械学習技術も見ていきますが、問題によってはより良い結果が得られる可能性があるからです。

We will look at how we should preprocess and clean the data as well as how to interpret the result of our models after training, but first, we will see how we can feed columns that contain categories into a model that expects numbers by using embeddings.

データの前処理やクリーニングの方法、学習後のモデルの結果の見方などについても見ていきますが、まずは埋め込みを利用して、カテゴリを含む列を数字を想定したモデルに送り込む方法について見ていきます。

Categorical Embeddings

カテゴリの埋め込み

In tabular data some columns may contain numerical data, like "age," while others contain string values, like "sex." The numerical data can be directly fed to the model (with some optional preprocessing), but the other columns need to be converted to numbers. Since the values in those correspond to different categories, we often call this type of

variables *categorical variables*. The first type are called *continuous variables*.

表形式のデータでは、"年齢"のような数値データを含む列もあれば、"性別"のような文字列を含む列もある。数値データは（オプションで前処理をすれば）直接モデルに与えることができますが、それ以外の列は数値に変換する必要があります。これらの値は異なるカテゴリーに対応するため、このタイプの変数をカテゴリー変数と呼ぶことが多い。最初のタイプは連続変数と呼ばれる。

jargon: Continuous and Categorical Variables: Continuous variables are numerical data, such as "age," that can be directly fed to the model, since you can add and multiply them directly. Categorical variables contain a number of discrete levels, such as "movie ID," for which addition and multiplication don't have meaning (even if they're stored as numbers).

専門用語：連続変数とカテゴリー変数：連続変数は、「年齢」のような数値データで、直接足したり掛けたりすることができますので、直接モデルに与えることができます。カテゴリー変数は、「映画の ID」のように離散的なレベルをいくつも含み、足し算や掛け算が（数字として保存されていても）意味をなさない。

At the end of 2015, the [Rossmann sales competition](#) ran on Kaggle. Competitors were given a wide range of information about various stores in Germany, and were tasked with trying to predict sales on a number of days. The goal was to help the company to manage stock properly and be able to satisfy demand without holding unnecessary inventory. The official training set provided a lot of information about the stores. It was also permitted for competitors to use additional data, as long as that data was made public and available to all participants.

2015年末、Kaggle でロスマンセールスコンペティションが実施された。競技者は、ドイツのさまざまな店舗に関するさまざまな情報を与えられ、何日かの売上を予測することに挑戦することになった。その目的は、会社が適切に在庫を管理し、不必要な在庫を持たずに需要を満たすことができるようになります。公式トレーニングセットでは、店舗に関する多くの情報が提供されました。また、そのデータが公開され、すべての参加者が利用できるのであれば、競技者が追加のデータを使用することも許可されました。

One of the gold medalists used deep learning, in one of the earliest known examples of a state-of-the-art deep learning tabular model. Their method involved far less feature engineering, based on domain knowledge, than those of the other gold medalists. The paper, "[Entity Embeddings of Categorical Variables](#)" describes their approach. In an online-only chapter on the [book's website](#) we show how to replicate it from scratch and attain the same accuracy shown in the paper. In the abstract of the paper the authors (Cheng Guo and Felix Berkhahn) say: 金メダリストの1人はディープラーニングを使用し、最先端のディープラーニングの表形式モデルの最も初期の例として知られています。その方法は、他の金メダリストの方法と比べて、ドメイン知識に基づく特徴工学をはるかに少なくしています。論文「[Entity Embeddings of Categorical Variables](#)」は、彼らのアプローチについて説明しています。

本書のウェブサイトにあるオンライン限定の章では、この論文をゼロから再現し、論文で示されたのと同じ精度を達成する方法を紹介しています。論文の要旨で著者（Cheng Guo と Felix Berkhahn）はこう言っています：

: Entity embedding not only reduces memory usage and speeds up neural networks compared with one-hot encoding, but more importantly by mapping similar values close to each other in the embedding space it reveals the intrinsic properties of the categorical variables... [It] is especially useful for datasets with lots of high cardinality features, where other methods tend to overfit... As entity embedding defines a distance measure for categorical variables it can be used for visualizing categorical data and for data clustering.

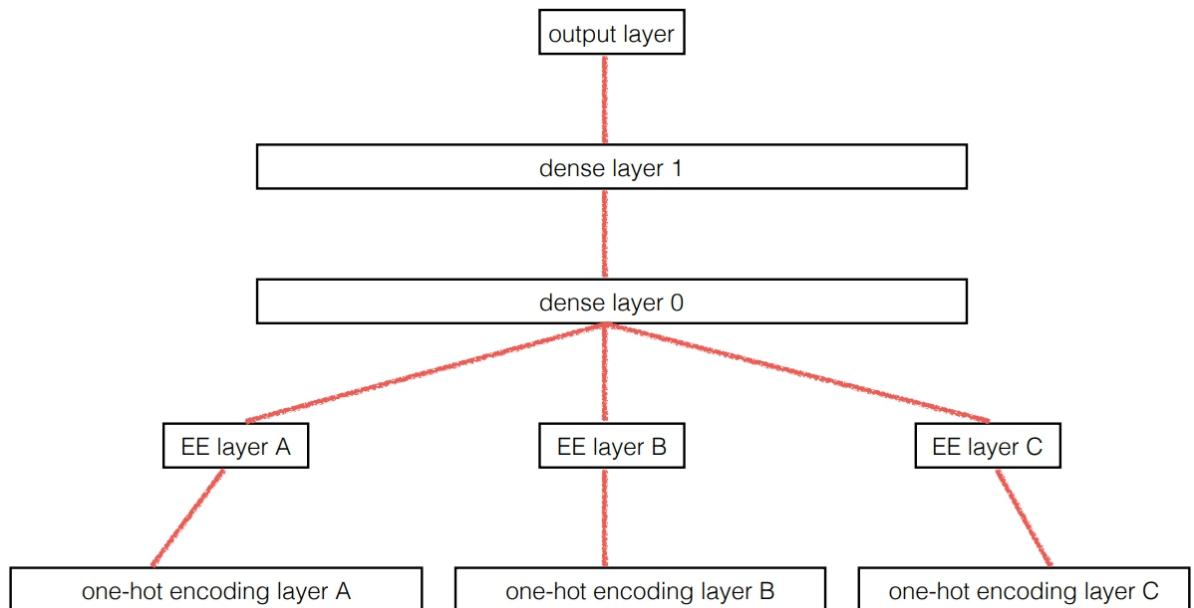
：エンティティエンベッディングは、ワンホットエンコーディングと比較して、メモリ使用量を削減し、ニューラルネットワークを高速化するだけでなく、エンベッディング空間において類似の値を互いに近づけることによって、カテゴリ変数の本質的な特性を明らかにすることができます。[他の手法ではオーバーフィットしがちな、カーディナリティの高い特徴を多く含むデータセットに特に有効である… エンティティエンベッディングは、カテゴリ変数の距離尺度を定義しているので、カテゴリデータの可視化やデータのクラスタリングに利用できる。

We have already noticed all of these points when we built our collaborative filtering model. We can clearly see that these insights go far beyond just collaborative filtering, however.

これらの点は、協調フィルタリングモデルを構築する際にすでに気づいていたことです。しかし、これらの知見は、単なる協調フィルタリングにとどまらないことがよくわかる。

The paper also points out that (as we discussed in the last chapter) an embedding layer is exactly equivalent to placing an ordinary linear layer after every one-hot-encoded input layer. The authors used the diagram in <>entity_emb> to show this equivalence. Note that "dense layer" is a term with the same meaning as "linear layer," and the one-hot encoding layers represent inputs.

また、この論文では、（前章で述べたように）埋め込み層は、1ホットエンコードされた入力層の後に通常の線形層を配置することと全く同じであることを指摘している。著者らはこの等価性を示すために<>の図を使っている。なお、「密層」は「線形層」と同じ意味の用語であり、1ホットエンコード層は入力を表しています。



The insight is important because we already know how to train linear layers, so this shows that from the point of view of the architecture and our training algorithm the embedding layer is just another layer. We also saw this in practice in the last chapter, when we built a collaborative filtering neural network that looks exactly like this diagram.

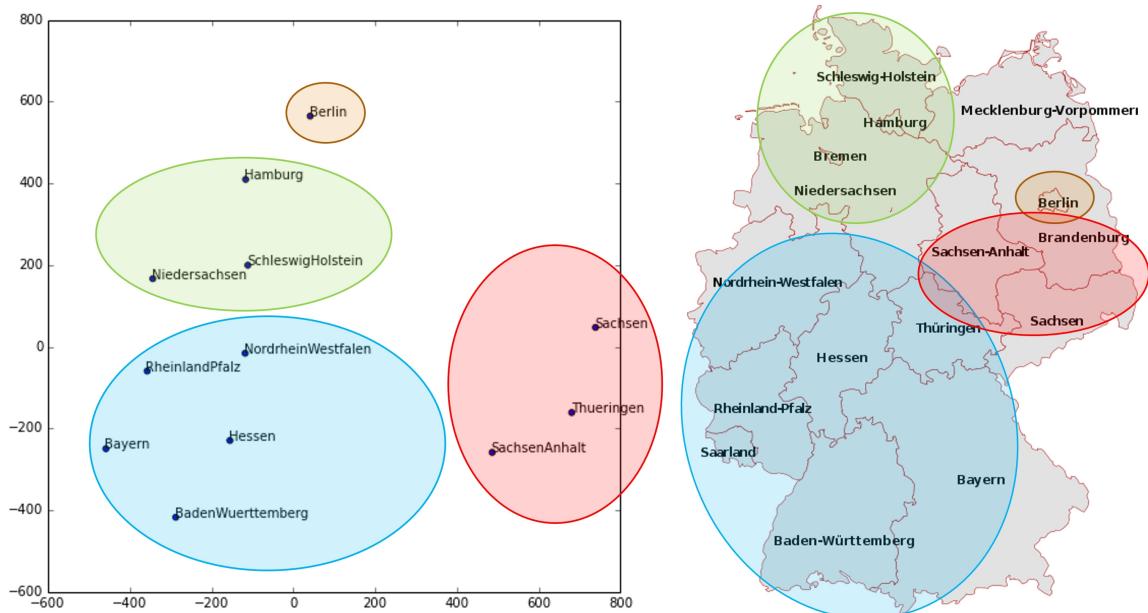
ニューラルネットワークにおけるエンティティの埋め込み
この洞察は重要です。なぜなら、私たちはすでに線形層の学習方法を知っているからです。
このことは、アーキテクチャと学習アルゴリズムの観点からは、埋め込み層は単なる別の層であることを示します。前章では、この図のような協調フィルタリングニューラルネットワークを構築し、その実践を行いました。

Where we analyzed the embedding weights for movie reviews, the authors of the entity embeddings paper analyzed the embedding weights for their sales prediction model. What they found was quite amazing, and illustrates their second key insight. This is that the embedding transforms the categorical variables into inputs that are both continuous and meaningful.

私たちが映画レビューの埋め込み重みを分析したように、entity embeddings の論文の著者は、売上予測モデルの埋め込み重みを分析しました。この結果は、非常に驚くべきものであり、彼らの 2 つ目の重要な洞察を示している。それは、エンベッディングがカテゴリ変数を連続的で意味のある入力に変換することです。

The images in <<state_emb>> illustrate these ideas. They are based on the approaches used in the paper, along with some analysis we have added.

<>の画像は、これらのアイデアを説明するものです。これらは、論文で使用されたアプローチに基づいており、私たちが追加した分析も含まれています。

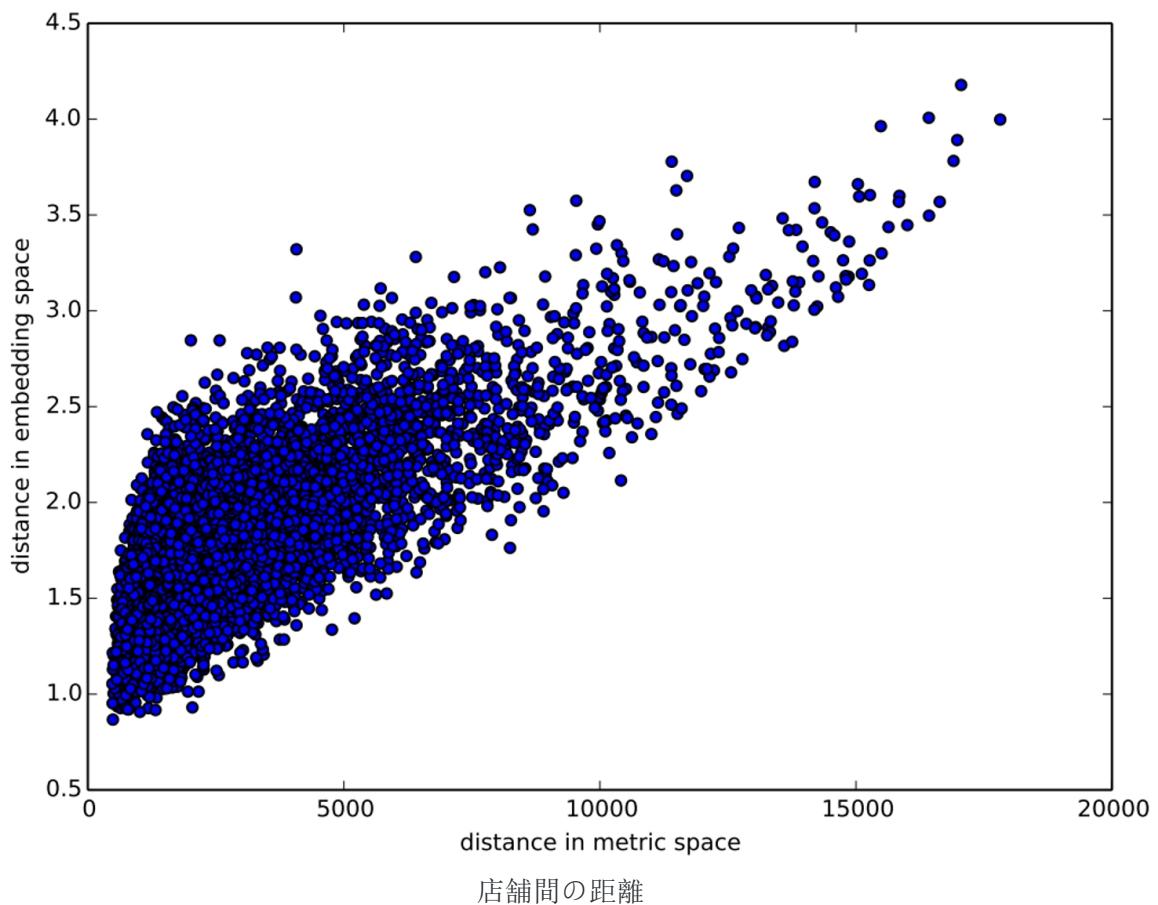


On the left is a plot of the embedding matrix for the possible values of the State category. For a categorical variable we call the possible values of the variable its "levels" (or "categories" or "classes"), so here one level is "Berlin," another is "Hamburg," etc. On the right is a map of Germany. The actual physical locations of the German states were not part of the provided data, yet the model itself learned where they must be, based only on the behavior of store sales!

左側は、State のカテゴリの可能な値に対する埋め込み行列のプロットです。カテゴリ変数では、変数の可能な値を「レベル」（または「カテゴリ」「クラス」）と呼びますので、ここでは、あるレベルは「ベルリン」、別のレベルは「ハンブルク」などです。右側はドイツの地図です。ドイツ各州の実際の位置は、提供されたデータには含まれていませんが、モデル自身が、店舗の販売行動だけに基づいて、その位置を学習しています！

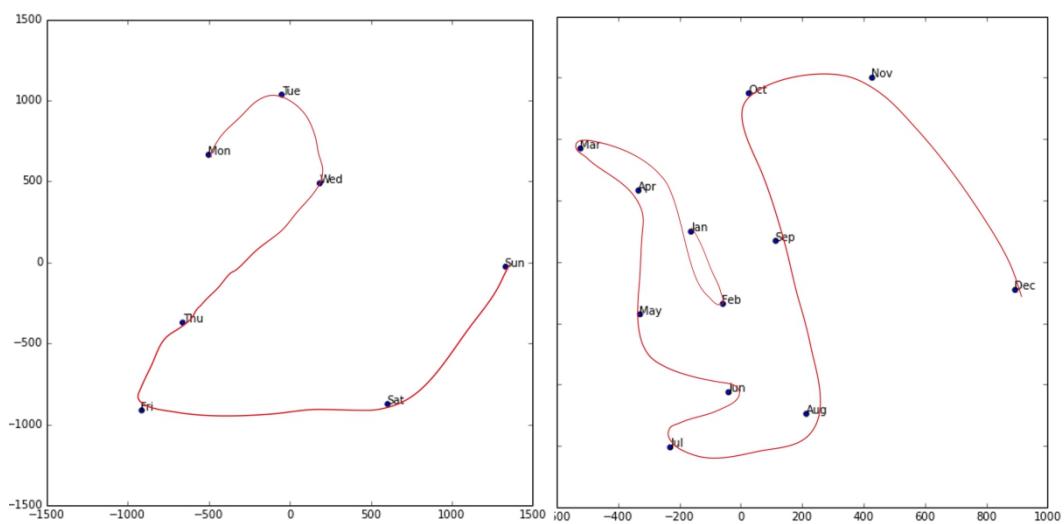
Do you remember how we talked about *distance* between embeddings? The authors of the paper plotted the distance between store embeddings against the actual geographic distance between the stores (see <>store_emb>>). They found that they matched very closely!

埋め込み間の距離の話をしたのを覚えてますか？この論文の著者は、店舗埋め込み間の距離を、実際の店舗間の地理的距離（<>参照）にプロットしました。その結果、両者は非常によく一致することがわかりました！



We've even tried plotting the embeddings for days of the week and months of the year, and found that days and months that are near each other on the calendar ended up close as embeddings too, as shown in <>date_emb>>.

さらに、曜日や月の埋め込みをプロットしてみると、<>のように、カレンダー上で近い日や月も埋め込みとして近くなってしまうことがわかりました。



What stands out in these two examples is that we provide the model fundamentally categorical data about discrete entities (e.g., German states or days of the week), and then the model learns an embedding for these entities that defines a continuous notion of distance between them. Because the embedding distance was learned based on real patterns in the data, that distance tends to match up with our intuitions.

この 2 つの例で顕著なのは、離散的なエンティティ（例えば、ドイツの州や曜日）に関する基本的なカテゴリーデータをモデルに与え、そのエンティティ間の距離の連続概念を定義する埋め込みをモデルが学習していることです。埋め込み距離は、データの実際のパターンに基づいて学習されているため、その距離は私たちの直感と一致する傾向がある。

In addition, it is valuable in its own right that embeddings are continuous, because models are better at understanding continuous variables. This is unsurprising considering models are built of many continuous parameter weights and continuous activation values, which are updated via gradient descent (a learning algorithm for finding the minimums of continuous functions).

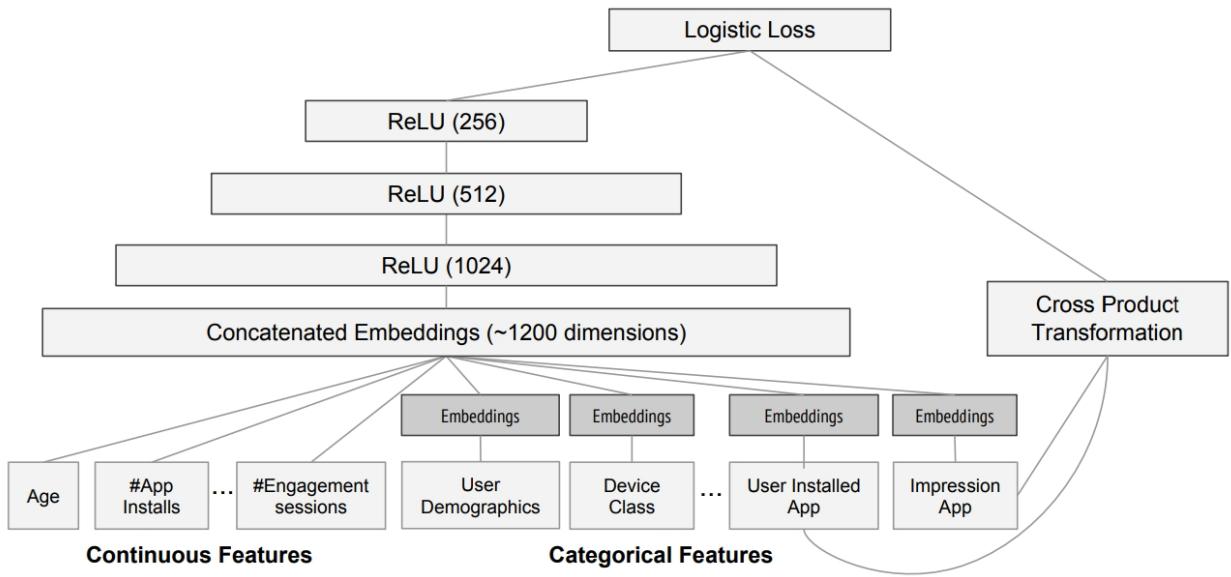
また、埋め込み距離が連続的であることは、それだけで価値があります。なぜなら、モデルは連続的な変数を理解するのに適しているからです。これは、モデルが多くの連続的なパラメータ重みと連続的な活性化値で構成され、勾配降下（連続関数の最小値を求める学習アルゴリズム）によって更新されることを考えれば、当然といえば当然である。

Another benefit is that we can combine our continuous embedding values with truly continuous input data in a straightforward manner: we just concatenate the variables, and feed the concatenation into our first dense layer. In other words, the raw categorical data is transformed by an embedding layer before it interacts with the raw continuous input data. This is how fastai and Guo and Berkahn handle tabular models containing continuous and categorical variables.

もう一つの利点は、連続的な埋め込み値と、本当に連続的な入力データを、簡単な方法で組み合わせることができることだ。変数を連結し、その連結を最初の密な層に送り込むだけである。言い換えれば、生のカテゴリーデータは、生の連続入力データと相互作用する前に、埋め込み層によって変換されます。これは、fastai や Guo and Berkahn が、連続変数とカテゴリーベクトルを含む表形式モデルを扱う方法です。

An example using this concatenation approach is how Google does its recommendations on Google Play, as explained in the paper "[Wide & Deep Learning for Recommender Systems](#)". <>[google_recsys](#)>> illustrates.

この連結アプローチを用いた例として、論文「Wide & Deep Learning for Recommender Systems」で説明されているように、Google が Google Play でレコメンデーションを行う方法がある。<>で説明されています。



Google Play のレコメンデーションシステム

Interestingly, the Google team actually combined both approaches we saw in the previous chapter: the dot product (which they call *cross product*) and neural network approaches.

興味深いことに、Google チームは、前章で見たドットプロダクト（彼らはクロスプロダクトと呼んでいる）とニューラルネットワークのアプローチという 2 つのアプローチを実際に組み合わせている。

Let's pause for a moment. So far, the solution to all of our modeling problems has been: *train a deep learning model*. And indeed, that is a pretty good rule of thumb for complex unstructured data like images, sounds, natural language text, and so forth. Deep learning also works very well for collaborative filtering. But it is not always the best starting point for analyzing tabular data.

ちょっと立ち止まってみましょう。これまでのところ、モデリングに関するすべての問題の解決策は、「ディープラーニングモデルをトレーニングする」というものでした。そして実際、画像、音声、自然言語テキストなどの複雑な非構造化データに対しては、これはかなり良い経験則です。また、ディープラーニングは協調フィルタリングにも非常に有効です。しかし、表形式のデータを分析する際には、必ずしも最適な出発点とは言えません。

Beyond Deep Learning

ディープラーニングを超えて

Most machine learning courses will throw dozens of different algorithms at you, with a brief technical description of the math behind them and maybe a toy example. You're left confused by the enormous range of techniques shown and have little practical understanding of how to apply them.

ほとんどの機械学習コースでは、何十種類ものアルゴリズムが、その背後にある数学の簡単な技術的説明と、おそらくおもちゃの例とともに、あなたに投げかけられます。そして、その背後にある数学の簡単な技術的な説明と模範解答が示されるだけです。

The good news is that modern machine learning can be distilled down to a couple of key techniques that are widely applicable. Recent studies have shown that the vast majority of datasets can be best modeled with just two methods:

しかし、現代の機械学習は、広く応用可能ないくつかの重要な技術に絞ることができるのです。最近の研究では、大多数のデータセットが、たった2つの手法で最適にモデル化できることが示されている：

1. Ensembles of decision trees (i.e., random forests and gradient boosting machines), mainly for structured data (such as you might find in a database table at most companies)

決定木のアンサンブル（ランダムフォレストやグラディエントブースティングマシンなど）：主に構造化データ（多くの企業でデータベースのテーブルにあるようなもの）を対象とする

2. Multilayered neural networks learned with SGD (i.e., shallow and/or deep learning), mainly for unstructured data (such as audio, images, and natural language)

主に非構造化データ（音声、画像、自然言語など）を対象とした SGD（浅い学習、深い学習）で学習した多層ニューラルネットワーク

Although deep learning is nearly always clearly superior for unstructured data, these two approaches tend to give quite similar results for many kinds of structured data. But ensembles of decision trees tend to train faster, are often easier to interpret, do not require special GPU hardware for inference at scale, and often require less hyperparameter tuning. They have also been popular for quite a lot longer than deep learning, so there is a more mature ecosystem of tooling and documentation around them.

非構造化データではディープラーニングの方が明らかに優れていることがほとんどですが、多くの種類の構造化データでは、この2つのアプローチは非常に似た結果を出す傾向があります。しかし、決定木のアンサンブルは、学習速度が速く、解釈が容易で、大規模な推論に特別な GPU ハードウェアを必要とせず、ハイパーパラメータのチューニングもあり必要ない場合が多い。また、決定木は深層学習よりもずっと前から普及しているため、ツールやドキュメントなどのエコシステムがより成熟しています。

Most importantly, the critical step of interpreting a model of tabular data is significantly easier for decision tree ensembles. There are tools and methods for answering the pertinent questions, like: Which columns in the dataset were the most important for your predictions? How are they related to the dependent variable? How do they interact with each other? And which particular features were most important for some particular observation?

最も重要なのは、表形式データのモデルを解釈するという重要なステップが、決定木アンサンブルの方がはるかに簡単であるということです。以下のような適切な質問に答えるためのツールや方法があります： データセットのどの列が予測に最も重要だったのか？ それ

らは従属変数とどのように関連しているのか？どのように相互作用しているのか？また、ある特定の観測に対して、どの特定の特徴が最も重要であったか？

Therefore, ensembles of decision trees are our first approach for analyzing a new tabular dataset.

したがって、決定木のアンサンブルは、新しい表形式のデータセットを分析するための最初のアプローチとなります。

The exception to this guideline is when the dataset meets one of these conditions:

このガイドラインの例外は、データセットがこれらの条件のいずれかを満たしている場合である：

- There are some high-cardinality categorical variables that are very important ("cardinality" refers to the number of discrete levels representing categories, so a high-cardinality categorical variable is something like a zip code, which can take on thousands of possible levels).

非常に重要な高基準のカテゴリー変数がある（「基数」はカテゴリーを表す離散的なレベルの数を指すので、高基準のカテゴリー変数とは郵便番号のようなもので、何千ものレベルが可能である）。

- There are some columns that contain data that would be best understood with a neural network, such as plain text data.

プレーンテキストデータのように、ニューラルネットワークで理解するのが最適なデータを含むカラムも存在する。

In practice, when we deal with datasets that meet these exceptional conditions, we always try both decision tree ensembles and deep learning to see which works best. It is likely that deep learning will be a useful approach in our example of collaborative filtering, as we have at least two high-cardinality categorical variables: the users and the movies. But in practice things tend to be less cut-and-dried, and there will often be a mixture of high- and low-cardinality categorical variables and continuous variables.

実際には、このような例外的な条件を満たすデータセットを扱うときは、必ず決定木アンサンブルと深層学習の両方を試して、どちらが最適かを確認します。今回の協調フィルタリングの例では、ユーザーと映画という少なくとも2つの高基準のカテゴリー変数があるため、深層学習が有効なアプローチになる可能性があります。しかし、実際のところ、物事はそれほど単純ではなく、高カーディナリティと低カーディナリティのカテゴリー変数と連続変数が混在していることが多い。

Either way, it's clear that we are going to need to add decision tree ensembles to our modeling toolbox!

いずれにせよ、決定木アンサンブルをモデリングツールボックスに追加する必要があることは明らかです！

Up to now we've used PyTorch and fastai for pretty much all of our heavy lifting. But these libraries are mainly designed for algorithms that do lots of matrix multiplication and derivatives (that is, stuff like deep learning!). Decision trees don't depend on these operations at all, so PyTorch isn't much use.

これまで私たちは、PyTorch と fastai を使って、ほとんどすべての作業をこなしてきました。しかし、これらのライブラリは主に行列の乗算や微分を多用するアルゴリズム（つまり、ディープラーニングのようなもの！）用に設計されています。決定木はこれらの演算に全く依存しないので、PyTorch はあまり役に立ちません。

Instead, we will be largely relying on a library called scikit-learn (also known as sklearn). Scikit-learn is a popular library for creating machine learning models, using approaches that are not covered by deep learning. In addition, we'll need to do some tabular data processing and querying, so we'll want to use the Pandas library. Finally, we'll also need NumPy, since that's the main numeric programming library that both sklearn and Pandas rely on.

その代わりに、scikit-learn（別名 sklearn）と呼ばれるライブラリに大きく依存することになります。scikit-learn は機械学習モデルを作成するための一般的なライブラリで、ディープラーニングではカバーできないアプローチを使っています。さらに、表形式のデータ処理とクエリーを行う必要があるので、Pandas ライブラリを使用します。最後に、sklearn と Pandas の両方が依存する主要な数値プログラミング・ライブラリである NumPy も必要です。

We don't have time to do a deep dive into all these libraries in this book, so we'll just be touching on some of the main parts of each. For a far more in depth discussion, we strongly suggest Wes McKinney's [Python for Data Analysis](#) (O'Reilly). Wes is the creator of Pandas, so you can be sure that the information is accurate!

この本では、これらすべてのライブラリを深く掘り下げる時間がないので、それぞれの主要な部分について触れるだけにしておきます。もっと深く知りたい方は、Wes McKinney の「Python for Data Analysis」（O'Reilly）を強くお勧めします。Wes は Pandas の生みの親なので、情報が正確であることは間違いないでしょう！

First, let's gather the data we will use.

まず、使用するデータを集めましょう。

The Dataset

データセット

The dataset we use in this chapter is from the Blue Book for Bulldozers Kaggle competition, which has the following description: "The goal of the contest is to predict the sale price of a particular piece of heavy equipment at auction based on its usage, equipment type, and configuration. The data is sourced from auction result postings and includes information on usage and equipment configurations."

本章で使用するデータセットは、Blue Book for Bulldozers という Kaggle コンペティションのもので、以下の説明があります： "コンテストの目的は、オークションにおける特定の重機の売却価格を、その使用状況、機器の種類、構成に基づいて予測することである。デ

ータはオークション結果の投稿から入手し、使用状況や機器の構成に関する情報を含んでいます。"

This is a very common type of dataset and prediction problem, similar to what you may see in your project or workplace. The dataset is available for download on Kaggle, a website that hosts data science competitions.

これは、あなたのプロジェクトや職場で見られるような、非常に一般的なタイプのデータセットと予測問題です。このデータセットは、データサイエンスのコンペティションを開催しているウェブサイト「Kaggle」でダウンロード可能です。

Kaggle Competitions

Kaggle コンペティション

Kaggle is an awesome resource for aspiring data scientists or anyone looking to improve their machine learning skills. There is nothing like getting hands-on practice and receiving real-time feedback to help you improve your skills.

Kaggle は、データサイエンティスト志望者や機械学習のスキルアップを目指す人にとって、素晴らしいリソースです。実践的な練習とリアルタイムのフィードバックは、スキルアップに役立つことこの上ないです。

Kaggle provides:

Kaggle は以下を提供します:

- Interesting datasets
興味深いデータセット
- Feedback on how you're doing
自分の状態をフィードバックする
- A leaderboard to see what's good, what's possible, and what's state-of-the-art
何が良いのか、何が可能なのか、何が最先端なのかを確認できるリーダーボード
- Blog posts by winning contestants sharing useful tips and techniques
入賞者のブログで、役立つヒントやテクニックを紹介しています。

Until now all our datasets have been available to download through fastai's integrated dataset system. However, the dataset we will be using in this chapter is only available from Kaggle. Therefore, you will need to register on the site, then go to the [page for the competition](#). On that page click "Rules," then "I Understand and Accept." (Although the competition has finished, and you will not be entering it, you still have to agree to the rules to be allowed to download the data.)

これまで、私たちのデータセットはすべて fastai の統合データセットシステムからダウンロードすることができました。しかし、本章で使用するデータセットは、Kaggle からしか入手できません。そのため、Kaggle に登録し、コンペティションのページに移動する必要があります。そのページで "Rules"、"I Understand and Accept" の順にクリックします。(コンペティションは終了しており、あなたがエントリーすることはできませんが、データ

のダウンロードを許可されるためには、ルールに同意する必要があります)

The easiest way to download Kaggle datasets is to use the Kaggle API. You can install this using pip by running this in a notebook cell:

Kaggle のデータセットをダウンロードする最も簡単な方法は、Kaggle API を使用することです。ノートブックセルでこれを実行すると、pip を使ってインストールすることができます：

```
!pip install kaggle
```

You need an API key to use the Kaggle API; to get one, click on your profile picture on the Kaggle website, and choose My Account, then click Create New API Token. This will save a file called *kaggle.json* to your PC. You need to copy this key on your GPU server. To do so, open the file you downloaded, copy the contents, and paste them in the following cell in the notebook associated with this chapter (e.g., creds = '{"username": "xxx", "key": "xxx"}'):

Kaggle API を使用するには API キーが必要です。API キーを取得するには、Kaggle ウェブサイトでプロフィール画像をクリックし、My Account を選択し、Create New API Token をクリックしてください。これにより、*kaggle.json* というファイルがあなたの PC に保存されます。このキーを GPU サーバーにコピーする必要があります。そのためには、ダウンロードしたファイルを開いて内容をコピーし、この章に関連するノートブックの以下のセルに貼り付けます（例：creds = '{"username": "xxx", "key": "xxx"}'）：

In []:

```
creds = "
```

Then execute this cell (this only needs to be run once):

次に、このセルを実行します（これは一度だけ実行すればよい）：

In []:

```
cred_path = Path('~/kaggle/kaggle.json').expanduser()
if not cred_path.exists():
    cred_path.parent.mkdir(exist_ok=True)
    cred_path.write_text(creds)
    cred_path.chmod(0o600)
```

Now you can download datasets from Kaggle! Pick a path to download the dataset to:

Kaggle からデータセットをダウンロードできるようになりました！データセットをダウンロードするパスを選んでください：

In []:

```
comp = 'bluebook-for-bulldozers' path = URLs.path(comp) path
```

Out[]:

```
Path('/home/jhoward/.fastai/archive/bluebook-for-bulldozers')
```

In []:

```
#hide
```

```
Path.BASE_PATH = path
```

And use the Kaggle API to download the dataset to that path, and extract it:

そして、Kaggle API を使って、そのパスにデータセットをダウンロードし、抽出します：

In []:

```
from kaggle import api
if not path.exists():
    path.mkdir(parents=true)
    api.competition_download_cli(comp, path=path)
    shutil.unpack_archive(str(path/f'{comp}.zip'), str(path))
path.ls(file_type='text')
```

Out[]:

```
(#7) [Path('ValidSolution.csv'), Path('Machine_Appendix.csv'), Path('TrainAn
dValid.csv'), Path('median_benchmark.csv'), Path('random_forest_benchmark
_test.csv'), Path('Test.csv'), Path('Valid.csv')]
```

Now that we have downloaded our dataset, let's take a look at it!

さて、データセットのダウンロードが完了しましたので、見てみましょう！

Look at the Data

データを見る

Kaggle provides information about some of the fields of our dataset. The [Data](#) explains that the key fields in *train.csv* are:

Kaggle は、私たちのデータセットのいくつかのフィールドに関する情報を提供しています。データでは、*train.csv* の主要なフィールドは以下の通りだと説明されています：

- SalesID:: The unique identifier of the sale.
SalesID:: 販売の一意な識別子。
- MachineID:: The unique identifier of a machine. A machine can be sold multiple times.
MachineID::マシンの一意な識別子。1台のマシンは複数回販売することができる。
- saleprice:: What the machine sold for at auction (only provided in *train.csv*).
saleprice::オークションでマシンがいくらで売れたか (*train.csv* にのみ提供される)。

- saledate:: The date of the sale.

saledate:: 売却された日付。

In any sort of data science work, it's important to *look at your data directly* to make sure you understand the format, how it's stored, what types of values it holds, etc. Even if you've read a description of the data, the actual data may not be what you expect. We'll start by reading the training set into a Pandas DataFrame. Generally it's a good idea to specify low_memory=False unless Pandas actually runs out of memory and returns an error.

The low_memory parameter, which is True by default, tells Pandas to only look at a few rows of data at a time to figure out what type of data is in each column. This means that Pandas can actually end up using different data type for different rows, which generally leads to data processing errors or model training problems later.

データサイエンスの仕事では、データを直接見て、フォーマット、保存方法、保持する値の種類などを理解することが重要です。データの説明を読んだとしても、実際のデータは期待通りとは限りません。まずはトレーニングセットを Pandas DataFrame に読み込むところから始めます。一般的には、Pandas が実際にメモリ不足になってエラーを返さない限り、low_memory=False を指定するのがよいでしょう。low_memory パラメータはデフォルトで True になっており、Pandas に一度に数行のデータしか見ずに、各列にどんな種類のデータがあるのかを把握させるように指示します。つまり、Pandas は実際には異なる行に対して異なるデータ型を使用してしまう可能性があり、一般的にはデータ処理のエラー やモデルトレーニングの問題を後で引き起こすことになります。

Let's load our data and have a look at the columns:

データを読み込んで、列を見てみましょう：

```
In [ ]:  
df = pd.read_csv(path/"TrainAndValid.csv", low_memory=False)  
In [ ]:  
df.columns  
Out[ ]:  
Index(['SalesID', 'SalePrice', 'MachineID', 'ModelID', 'datasource',  
       'auctioneerID', 'YearMade', 'MachineHoursCurrentMeter', 'UsageB  
and',  
       'saledate', 'fiModelDesc', 'fiBaseModel', 'fiSecondaryDesc',  
       'fiModelSeries', 'fiModelDescriptor', 'ProductSize',  
       'fiProductClassDesc', 'state', 'ProductGroup', 'ProductGroupDesc',  
       'Drive_System', 'Enclosure', 'Forks', 'Pad_Type', 'Ride_Control',  
       'Stick', 'Transmission', 'Turbocharged', 'Blade_Extension',  
       'Blade_Width', 'Enclosure_Type', 'Engine_Horsepower', 'Hydraulics  
,  
       'Pushblock', 'Ripper', 'Scarifier', 'Tip_Control', 'Tire_Size',
```

```
'Coupler', 'Coupler_System', 'Grouser_Tracks', 'Hydraulics_Flow',
'Track_Type', 'Undercarriage_Pad_Width', 'Stick_Length', 'Thumb',
'Pattern_Changer', 'Grouser_Type', 'Backhoe_Mounting', 'Blade_Ty
pe',
'Travel_Controls', 'Differential_Type', 'Steering_Controls'],
dtype='object')
```

That's a lot of columns for us to look at! Try looking through the dataset to get a sense of what kind of information is in each one. We'll shortly see how to "zero in" on the most interesting bits.

たくさん のカラムを見ることが出来ますね！データセットを見て、それぞれのカラムにどんな情報が入っているのか感じてみてください。最も興味深い部分を「ゼロイン」する方法については、後ほど説明する。

At this point, a good next step is to handle *ordinal columns*. This refers to columns containing strings or similar, but where those strings have a natural ordering. For instance, here are the levels of ProductSize:

この時点で、次のステップとして、序列列を扱うのがよいでしょう。これは、文字列やそれに類するものを含む列のことで、これらの文字列が自然な順序を持つものである。たとえば、ProductSize のレベルは次のとおりです：

```
In [ ]:  
df['ProductSize'].unique()  
Out[ ]:  
array([nan, 'Medium', 'Small', 'Large / Medium', 'Mini', 'Large', 'Compact
'], dtype=object)
```

We can tell Pandas about a suitable ordering of these levels like so:

これらのレベルの適切な順序について、次のように Pandas に伝えることができます：

```
In [ ]:  
sizes = 'Large','Large / Medium','Medium','Small','Mini','Compact'  
In [ ]:  
df['ProductSize'] = df['ProductSize'].astype('category')  
df['ProductSize'].cat.set_categories(sizes, ordered=True, inplace=True)
```

The most important data column is the dependent variable—that is, the one we want to predict. Recall that a model's metric is a function that reflects how good the predictions are. It's important to note what metric is being used for a project. Generally, selecting the metric is an important part of the project setup. In many cases, choosing a good metric will require more than just selecting a variable that already exists. It is more like a design process. You should think carefully about which metric, or set of metrics, actually measures the notion of model quality that

matters to you. If no variable represents that metric, you should see if you can build the metric from the variables that are available.

最も重要なデータ列は、従属変数、つまり、予測したい変数です。モデルのメトリックは、予測の良し悪しを反映する関数であることを思い出してください。プロジェクトでどのようなメトリックが使用されているかに注意することは重要である。一般的に、メトリックを選択することは、プロジェクトのセットアップの重要な部分である。多くの場合、良いメトリックを選択するには、すでに存在する変数を選択するだけでは不十分です。どちらかというと、設計のようなものです。あなたにとって重要なモデル品質の概念を実際に測定するのは、どのメトリクス、またはメトリクスのセットなのかを慎重に考える必要があります。その指標を表す変数がない場合は、利用可能な変数からその指標を構築できるかどうかを確認する必要があります。

However, in this case Kaggle tells us what metric to use: root mean squared log error (RMSLE) between the actual and predicted auction prices. We need do only a small amount of processing to use this: we take the log of the prices, so that rmse of that value will give us what we ultimately need:

しかし、この場合、Kaggle は、実際のオークション価格と予測されたオークション価格の間の平均二乗対数誤差 (RMSLE) を使用するように指示します。価格の対数を取るので、その値の rmse で最終的に必要なものが得られます:

```
In [ ]:  
dep_var = 'SalePrice'  
In [ ]:  
df[dep_var] = np.log(df[dep_var])
```

We are now ready to explore our first machine learning algorithm for tabular data: decision trees.

表形式データに対する最初の機械学習アルゴリズムである決定木について説明する準備が整いました。

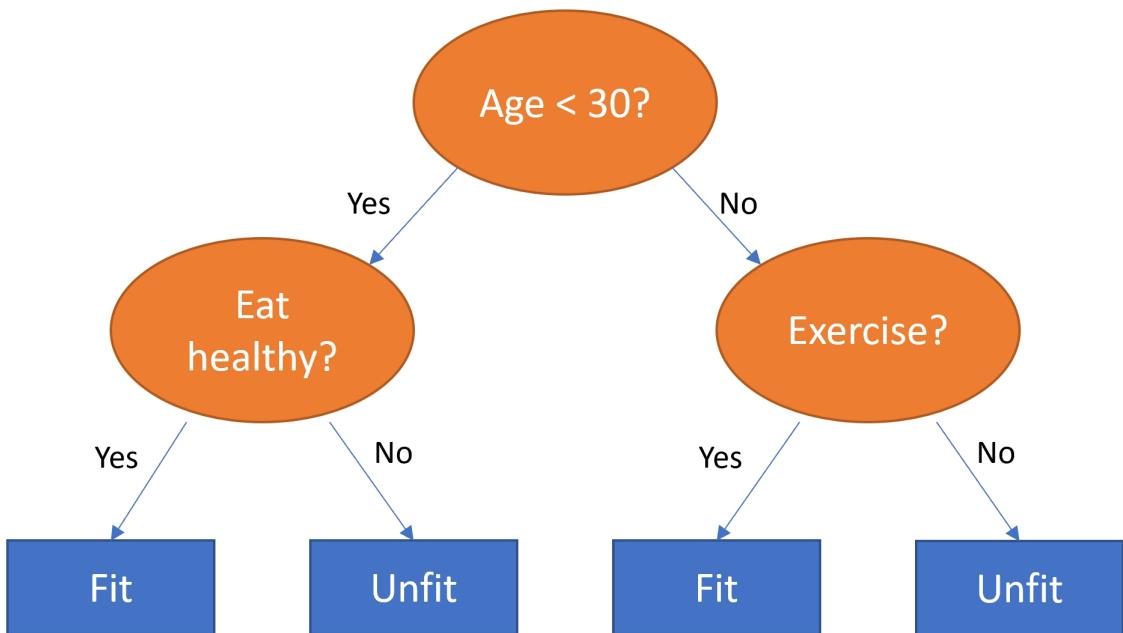
Decision Trees

決定木（デシジョンツリー）

Decision tree ensembles, as the name suggests, rely on decision trees. So let's start there! A decision tree asks a series of binary (that is, yes or no) questions about the data. After each question the data at that part of the tree is split between a "yes" and a "no" branch, as shown in <> decision_tree>. After one or more questions, either a prediction can be made on the basis of all previous answers or another question is required.

決定木アンサンブルは、その名の通り、決定木に依存するものです。そこで、まずそこから始めましょう！決定木は、データに対して一連のバイナリ（つまり、イエスかノーか）の質問をします。各質問の後、ツリーのその部分のデータは、<>に示すように、「はい」と「いいえ」の枝に分けられます。1つまたは複数の質問の後、以前のすべての回答に基づ

いて予測を行うことができるか、別の質問が必要です。



デシジョンツリーの一例

This sequence of questions is now a procedure for taking any data item, whether an item from the training set or a new one, and assigning that item to a group. Namely, after asking and answering the questions, we can say the item belongs to the same group as all the other training data items that yielded the same set of answers to the questions. But what good is this? The goal of our model is to predict values for items, not to assign them into groups from the training dataset. The value is that we can now assign a prediction value for each of these groups—for regression, we take the target mean of the items in the group.

この一連の質問は、トレーニングセットのアイテムであり、新しいアイテムであり、任意のデータアイテムを取り込み、そのアイテムをグループに割り当てるための手順となります。つまり、質問と回答の後、その項目は、質問に対して同じ回答を得た他のすべてのトレーニングデータと同じグループに属していると言うことができる。しかし、これは何の役に立つのでしょうか？私たちのモデルの目的は、アイテムの値を予測することであり、トレーニングデータセットからグループに割り当てるこではありません。価値あるのは、これらのグループのそれぞれについて予測値を割り当てるようになったことです。

Let's consider how we find the right questions to ask. Of course, we wouldn't want to have to create all these questions ourselves—that's what computers are for! The basic steps to train a decision tree can be written down very easily:

では、どのようにして適切な質問を見つけるかを考えてみましょう。もちろん、このような質問をすべて自分で作るのは面倒なので、コンピュータの出番です！決定木を学習させる基本的な手順は、非常に簡単に書き表すことができます：

1. Loop through each column of the dataset in turn.

データセットの各列を順番にループする。

2. For each column, loop through each possible level of that column in turn.

データセットの各列を順番にループする。各列について、その列の可能なレベルを順番にループする。

3. Try splitting the data into two groups, based on whether they are greater than or less than that value (or if it is a categorical variable, based on whether they are equal to or not equal to that level of that categorical variable).

データを2つのグループに分けて、その値より大きいか小さいかに基づいて（あるいは、それがカテゴリ変数であれば、そのカテゴリ変数のそのレベルと等しいか等しくないかに基づいて）、試してみる。

4. Find the average sale price for each of those two groups, and see how close that is to the actual sale price of each of the items of equipment in that group. That is, treat this as a very simple "model" where our predictions are simply the average sale price of the item's group.

その2つのグループのそれぞれの平均販売価格を求め、そのグループに属する各機器の実際の販売価格にどれだけ近いかを確認する。つまり、これは非常に単純な「モデル」として扱い、予測は単にアイテムのグループの平均販売価格とする。

5. After looping through all of the columns and all the possible levels for each, pick the split point that gave the best predictions using that simple model.

すべての列をループし、それに可能なすべてのレベルを設定した後、この単純なモデルを使って最も良い予測をした分割点を選びます。

6. We now have two different groups for our data, based on this selected split. Treat each of these as separate datasets, and find the best split for each by going back to step 1 for each group.

これで、選択した分割に基づく、2つの異なるグループのデータができました。それを別のデータセットとして扱い、各グループについてステップ1に戻って最適な分割点を探します。

7. Continue this process recursively, until you have reached some stopping criterion for each group—for instance, stop splitting a group further when it has only 20 items in it.

この処理を再帰的に行い、各グループで何らかの停止基準に達するまで続ける。例えば、あるグループのアイテム数が20個になったら、それ以上分割するのを止める。

Although this is an easy enough algorithm to implement yourself (and it is a good exercise to do so), we can save some time by using the implementation built into sklearn.

このアルゴリズムは自分で実装しても十分簡単ですが（実装するのも良い練習になります）、sklearnに組み込まれている実装を使うことで時間を節約することができます。

First, however, we need to do a little data preparation.

しかし、その前に少しデータを準備する必要があります。

A: Here's a productive question to ponder. If you consider that the procedure for defining a decision tree essentially chooses one *sequence of splitting questions about variables*, you might ask yourself, how do we know this procedure chooses the *correct sequence*? The rule is to choose the splitting question that produces the best split (i.e., that most accurately separates the items into two distinct categories), and then to apply the same rule to the groups that split produces, and so on. This is known in computer science as a "greedy" approach. Can you imagine a scenario in which asking a "less powerful" splitting question would enable a better split down the road (or should I say down the trunk!) and lead to a better result overall?

A: ここで、生産的な質問を考えてみましょう。決定木を定義する手順が、本質的に変数に関する分割質問の 1 つのシーケンスを選択することを考えると、この手順が正しいシーケンスを選択することをどうやって知ることができるのだろうかと思うかもしれません。そのルールとは、最も良い分割(すなわち、項目を 2 つの異なるカテゴリーに最も正確に分割すること)を生み出す分割質問を選び、その分割が生み出すグループにも同じルールを適用する、というものである。これはコンピュータサイエンスでは「貪欲な」アプローチとして知られています。あまり強力でない分割の質問をすることで、より良い分割が可能になり、全体としてより良い結果につながるというシナリオを想像できるでしょうか。

Handling Dates

取扱日

The first piece of data preparation we need to do is to enrich our representation of dates. The fundamental basis of the decision tree that we just described is *bisection*— dividing a group into two. We look at the ordinal variables and divide up the dataset based on whether the variable's value is greater (or lower) than a threshold, and we look at the categorical variables and divide up the dataset based on whether the variable's level is a particular level. So this algorithm has a way of dividing up the dataset based on both ordinal and categorical data.

データの準備としてまず必要なのは、日付の表現を豊かにすることです。先ほど説明した決定木の基本は、グループを 2 つに分ける「二分法」です。順序変数を見て、変数の値が閾値より大きい（または小さい）かどうかに基づいてデータセットを分割し、カテゴリ変数を見て、変数のレベルが特定のレベルであるかどうかに基づいてデータセットを分割し

ます。つまり、このアルゴリズムは、順序データとカテゴリーデータの両方に基づいてデータセットを分割する方法を持っているのです。

But how does this apply to a common data type, the date? You might want to treat a date as an ordinal value, because it is meaningful to say that one date is greater than another. However, dates are a bit different from most ordinal values in that some dates are qualitatively different from others in a way that that is often relevant to the systems we are modeling.

しかし、これを一般的なデータ型である日付に当てはめるとどうなるでしょうか。ある日付が他の日付より大きいというのは意味があるので、日付を順序値として扱いたいと思うかもしれません。しかし、日付は一般的な順序値とは少し異なり、ある日付は他の日付と質的に異なり、それは私たちがモデル化しているシステムにしばしば関連するものである。

In order to help our algorithm handle dates intelligently, we'd like our model to know more than whether a date is more recent or less recent than another. We might want our model to make decisions based on that date's day of the week, on whether a day is a holiday, on what month it is in, and so forth. To do this, we replace every date column with a set of date metadata columns, such as holiday, day of week, and month. These columns provide categorical data that we suspect will be useful.

アルゴリズムが日付を賢く扱えるようにするために、ある日付が他の日付より新しいか、新しいかだけでなく、モデルにも知っておいてもらいたいものです。例えば、日付の曜日、休日かどうか、何月か、などに基づいて判断するようにしたい。のために、日付の列を、休日、曜日、月といった日付のメタデータ列で置き換えます。これらのカラムは、役に立つと思われるカテゴリーデータを提供します。

fastai comes with a function that will do this for us—we just have to pass a column name that contains dates:

fastai には、これを行う関数が用意されており、日付を含むカラム名を渡すだけです:

In []:

```
df = add_datepart(df, 'saledate')
```

Let's do the same for the test set while we're there:

ついでにテストセットも同じようにしてみましょう:

In []:

```
df_test = pd.read_csv(path/'Test.csv', low_memory=False)
df_test = add_datepart(df_test, 'saledate')
```

We can see that there are now lots of new columns in our DataFrame:

DataFrame に新しいカラムがたくさん追加されたことが確認できます:

In []:

```
''.join(o for o in df.columns if o.startswith('sale'))
```

Out[]:

```
'saleWeek saleYear saleMonth saleDay saleDayofweek saleDayofyear sal  
eIs_month_end saleIs_month_start saleIs_quarter_end saleIs_quarter_sta  
rt saleIs_year_end saleIs_year_start saleElapsed'
```

This is a good first step, but we will need to do a bit more cleaning. For this, we will use fastai objects called TabularPandas and TabularProc.

これは良い最初のステップですが、もう少しクリーニングをする必要があります。そのために、TabularPandas と TabularProc と呼ばれる fastai オブジェクトを使用します。

Using TabularPandas and TabularProc

TabularPandas と TabularProc の使用について

A second piece of preparatory processing is to be sure we can handle strings and missing data. Out of the box, sklearn cannot do either. Instead we will use fastai's class TabularPandas, which wraps a Pandas DataFrame and provides a few conveniences. To populate a TabularPandas, we will use two TabularProcs, Categorify and FillMissing. A TabularProc is like a regular Transform, except that:

準備処理の 2 つ目は、文字列と欠損データを扱えることを確認することです。sklearn は、そのままではどちらもできません。代わりに、Pandas DataFrame をラップし、いくつかの便利な機能を提供する fastai のクラス TabularPandas を使用します。TabularPandas に入力するために、2 つの TabularProcs、Categorify と FillMissing を使用することにします。TabularProc は通常の Transform のようなものです：

- It returns the exact same object that's passed to it, after modifying the object in place.
渡されたオブジェクトと全く同じものを、その場で修正した上で返す。
- It runs the transform once, when data is first passed in, rather than lazily as the data is accessed.
データが最初に渡されたときに、データにアクセスするたびにダラダラと実行するのではなく、一度だけトランسفォームを実行します。

Categorify is a TabularProc that replaces a column with a numeric categorical column. FillMissing is a TabularProc that replaces missing values with the median of the column, and creates a new Boolean column that is set to True for any row where the value was missing. These two transforms are needed for nearly every tabular dataset you will use, so this is a good starting point for your data processing:

Categorify は TabularProc で、カラムを数値のカテゴリーカラムに置き換えます。FillMissing は TabularProc で、欠損値を列の中央値に置き換え、値が欠損している行に対して True に設定される新しい Boolean 列を作成します。この 2 つの変換は、使用するほぼすべての表形式データセットに必要なので、これはデータ処理の出発点として適しています：

In []:

```
procs = [Categorify, FillMissing]
```

TabularPandas will also handle splitting the dataset into training and validation sets for us. However we need to be very careful about our validation set. We want to design it so that it is like the *test set* Kaggle will use to judge the contest.

TabularPandas はデータセットをトレーニングセットとバリデーションセットに分割する処理も行ってくれます。しかし、検証セットには細心の注意を払う必要があります。Kaggle がコンテストの審査に使用するテストセットと同じように設計したいのです。

Recall the distinction between a validation set and a test set, as discussed in <<chapter_intro>>. A validation set is data we hold back from training in order to ensure that the training process does not overfit on the training data. A test set is data that is held back even more deeply, from us ourselves, in order to ensure that *we* don't overfit on the validation data, as we explore various model architectures and hyperparameters.

<>で説明した、検証セットとテストセットの区別を思い出してください。検証セットとは、学習データに対して学習プロセスがオーバーフィットしないことを確認するために、学習から遠ざかるデータのことです。テストセットは、様々なモデルアーキテクチャやハイパーパラメータを探索する際に、検証データでオーバーフィットしないようにするために、私たち自身がさらに深く制限されたデータです。

We don't get to see the test set. But we do want to define our validation data so that it has the same sort of relationship to the training data as the test set will have.

私たちはテストセットを見ることができません。しかし、検証データを定義することで、テストセットと同じような学習データとの関係性を持たせることができます。

In some cases, just randomly choosing a subset of your data points will do that. This is not one of those cases, because it is a time series.

場合によっては、データポイントのサブセットをランダムに選択するだけでも、それが可能です。しかし、このデータは時系列であるため、そのようなケースには当てはまりません。

If you look at the date range represented in the test set, you will discover that it covers a six-month period from May 2012, which is later in time than any date in the training set. This is a good design, because the competition sponsor will want to ensure that a model is able to predict the future. But it means that if we are going to have a useful validation set, we also want the validation set to be later in time than the training set. The Kaggle training data ends in April 2012, so we will define a narrower training dataset which consists only of the Kaggle training data from before November 2011, and we'll define a validation set consisting of data from after November 2011.

テストセットの日付範囲を見ると、2012年5月からの6ヶ月間であり、トレーニングセットのどの日付よりも時間的に遅れていることがわかります。コンペティションのスポンサーは、モデルが未来を予測できることを確認したいのですから、これは良い設計です。し

かし、これは、有用な検証セットを用意するのであれば、検証セットもトレーニングセットよりも遅い時期にする必要があることを意味します。Kaggle のトレーニングデータは 2012 年 4 月までなので、2011 年 11 月以前の Kaggle トレーニングデータのみからなる狭いトレーニングデータセットを定義し、2011 年 11 月以降のデータからなる検証セットを定義することにします。

To do this we use `np.where`, a useful function that returns (as the first element of a tuple) the indices of all True values:

そのために、タプルの最初の要素として、すべての True 値のインデックスを返す便利な関数である `np.where` を使用します：

In []:

```
cond = (df.saleYear<2011) | (df.saleMonth<10)
train_idx = np.where(cond)[0]
valid_idx = np.where(~cond)[0]
splits = (list(train_idx),list(valid_idx))
```

TabularPandas needs to be told which columns are continuous and which are categorical. We can handle that automatically using the helper function `cont_cat_split`:

TabularPandas は、どのカラムが連続的で、どのカラムがカテゴリー的であるかを教えてもらう必要があります。これはヘルパー関数 `cont_cat_split` を使って自動的に処理することができます：

In []:

```
cont,cat = cont_cat_split(df, 1, dep_var=dep_var)
```

In []:

```
to = TabularPandas(df, procs, cat, cont, y_names=dep_var, splits=splits)
```

A TabularPandas behaves a lot like a fastai Datasets object, including providing train and valid attributes:

TabularPandas は fastai Datasets オブジェクトと同じように動作し、`train` 属性と `valid` 属性を提供することができます：

In []:

```
len(to.train),len(to.valid)
```

Out[]:

```
(404710, 7988)
```

We can see that the data is still displayed as strings for categories (we only show a few columns here because the full table is too big to fit on a page):

データはまだカテゴリの文字列として表示されていることがわかります（全表は大きすぎてページに収まらないため、ここでは数列しか表示していません）：

In []:

#hide_output to.show(3)

saleWeek	UsageBand	fiModelDesc	fiBaseModel	fiSecondaryDesc	fiModelSeries	fiModelDescriptor	ProductSize	fiProductClassDesc	state	ProductGroup	
0	46	Low	521D	521	D	#na#	#na#	Wheel Loader - 110.0 to 120.0 Horsepower	Alabama	WL	
1	13	Low	950FII	950	F	II	#na# Medium	Wheel Loader - 150.0 to 175.0 Horsepower	North Carolina	WL	
2	9	High	226	226	#na#	#na#	#na# #na# 1351.0 to 1601.0 Lb Operating Capacity	Skid Steer Loader - New York	New York	SSL	
ProductGroupDesc	Drive_System	Enclosure	Forks	Pad_Type	Ride_Control	Stick	Transmission	Turbocharged	Blade_Extension	Blade_Width	Enclosure_Type
Wheel Loader	#na# EROPS w AC	None or Unspecified	#na#	None or Unspecified	#na#	#na#	#na#	#na#	#na#	#na#	#na#
Wheel Loader	#na# EROPS w AC	None or Unspecified	#na#	None or Unspecified	#na#	#na#	#na#	#na#	#na#	#na#	#na#
Skid Steer Loaders	#na# OROPS	None or Unspecified	#na#	#na# #na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#
Engine_Horsepower	Hydraulics	Pushblock	Ripper	Scarifier	Tip_Control	Tire_Size	Coupler	Coupler_System	Grouser_Tracks	Hydraulics_Flow	Track_Type
#na#	2 Valve	#na#	#na#	#na#	#na#	None or Unspecified	None or Unspecified	#na#	#na#	#na#	#na#
#na#	2 Valve	#na#	#na#	#na#	#na#	23.5	None or Unspecified	#na#	#na#	#na#	#na#
#na#	Auxiliary	#na#	#na#	#na#	#na#	#na#	None or Unspecified	None or Unspecified	None or Unspecified	Standard	#na#
Undercarriage_Pad_Width	Stick_Length	Thumb	Pattern_Changer	Grouser_Type	Backhoe_Mounting	Blade_Type	Travel_Controls	Differential_Type	Steering_Controls		
#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	Standard	Conventional	
#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	Standard	Conventional	
#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	#na#	
sales_month_end	sales_month_start	sales_quarter_end	sales_quarter_start	sales_year_end	sales_year_start	saleElapsed	auctioneerID_na				
False	False	False	False	False	False	False	1163635200	False			
False	False	False	False	False	False	False	1080259200	False			
False	False	False	False	False	False	False	1077753600	False			

MachineHoursCurrentMeter	saleYear	saleMonth	saleDay	saleDayofweek	saleDayofyear	SalePrice
68.0	2006	11	16	3	320	11.097410
4640.0	2004	3	26	4	86	10.950807
2838.0	2004	2	26	3	57	9.210340

#hide_input

```
to1 = TabularPandas(df, procs, ['state', 'ProductGroup', 'Drive_System', 'Enclosure'], [], y_names=dep_var,
splits=splits)
to1.show(3)
```

	state	ProductGroup	Drive_System	Enclosure	SalePrice
0	Alabama		WL	#na#	EROPS w AC 11.097410
1	North Carolina		WL	#na#	EROPS w AC 10.950807
2	New York		SSL	#na#	OROPS 9.210340

However, the underlying items are all numeric:

ただし、基礎となる項目はすべて数値です：

In []:

#hide_output

```
to.items.head(3)
```

Out[]:

	SalesID	SalePrice	MachineID	saleWeek	... saleIs_year_start	saleElapsed	auctioneerID_na	MachineHoursCurrentMeter_na
0	1139246	11.097410	999089	46	...	1	2647	1
1	1139248	10.950807	117657	13	...	1	2148	1
2	1139249	9.210340	434808	9	...	1	2131	1

3 rows × 67 columns

In []:

#hide_input

```
to1.items[['state', 'ProductGroup', 'Drive_System', 'Enclosure']].head(3)
```

Out[]:

	state	ProductGroup	Drive_System	Enclosure
0	1	6	0	3
1	33	6	0	3
2	32	3	0	6

The conversion of categorical columns to numbers is done by simply replacing each unique level with a number. The numbers associated with the levels are chosen consecutively as they are seen in a column, so there's no particular meaning to the numbers in categorical columns after conversion. The exception is if you first convert a column to a Pandas ordered category (as we did for ProductSize earlier), in which case the ordering you chose is used. We can see the mapping by looking at the classes attribute:

カテゴリカルカラムの数字への変換は、各ユニークなレベルを単に数字に置き換えることで行われます。レベルに関連する数字は、カラムで見られるように連続的に選択されるため、変換後のカテゴリカルカラムの数字に特別な意味はありません。例外は、最初にカラムを Pandas の順序付きカテゴリに変換する場合（先ほどの ProductSize のように）、その場合は選択した順序が使用されます。class 属性を見ることでマッピングを確認することができます：

```
In [ ]:  
to.classes['ProductSize']  
Out[ ]:  
['#na#', 'Large', 'Large / Medium', 'Medium', 'Small', 'Mini', 'Compact']
```

Since it takes a minute or so to process the data to get to this point, we should save it—that way in the future we can continue our work from here without rerunning the previous steps. fastai provides a save method that uses Python's *pickle* system to save nearly any Python object:

fastai は、Python の pickle システムを使って、ほぼすべての Python オブジェクトを保存する save メソッドを提供しています：

```
In [ ]:  
save_pickle(path/'to.pkl',to)
```

To read this back later, you would type:

これを後で読み返すには、次のように入力します：

```
to = (path/'to.pkl').load()
```

Now that all this preprocessing is done, we are ready to create a decision tree.

このような前処理が終わったので、いよいよ決定木の作成に入ります。

Creating the Decision Tree

デシジョンツリーを作成する

To begin, we define our independent and dependent variables:

はじめに、独立変数と従属変数を定義します：

In []:

```
#hide  
to = load_pickle(path/'to.pkl')  
  
xs,y = to.train.xs,to.train.y valid_xs,valid_y = to.valid.xs,to.valid.y
```

In []:

Now that our data is all numeric, and there are no missing values, we can create a decision tree:

データがすべて数値で、欠損値もないで、決定木を作成することができます：

In []:

```
m = DecisionTreeRegressor(max_leaf_nodes=4)  
m.fit(xs, y);
```

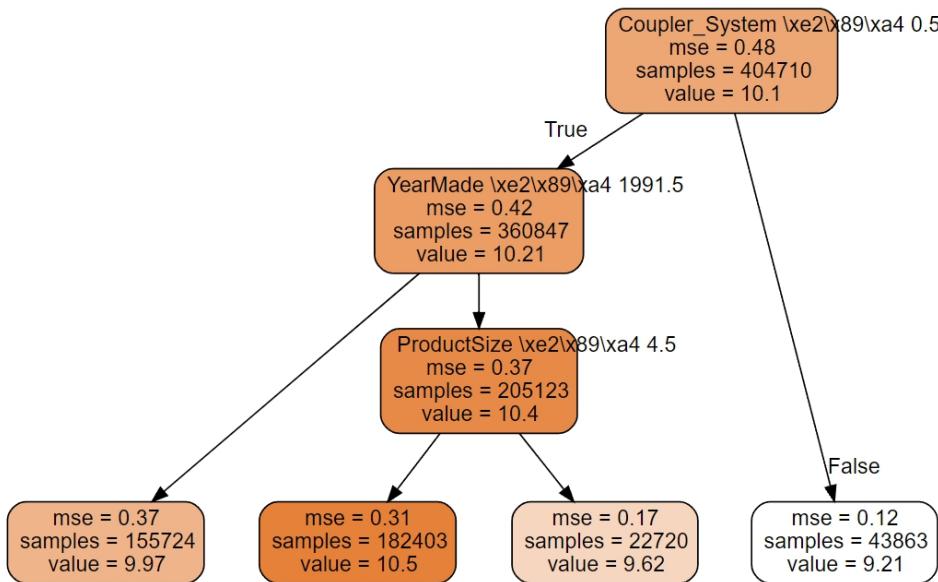
To keep it simple, we've told sklearn to just create four *leaf nodes*. To see what it's learned, we can display the tree:

シンプルにするために、sklearn には 4 つのリーフノードを作成するように指示しました。
何を学習したかを見るために、木を表示することができます：

In []:

```
draw_tree(m, xs, size=10, leaves_parallel=True, precision=2)
```

Out[]:



Understanding this picture is one of the best ways to understand decision trees, so we will start at the top and explain each part step by step.

この図を理解することは、決定木を理解するための最良の方法の 1 つであるため、上から順番に各部を説明していきます。

The top node represents the *initial model* before any splits have been done, when all the data is in one group. This is the simplest possible model. It is the result of asking zero questions and will always predict the value to be the average value of the whole dataset. In this case, we can see it predicts a value of 10.10 for the logarithm of the sales price. It gives a mean squared error of 0.48. The square root of this is 0.69. (Remember that unless you see `m_rmse`, or a *root mean squared error*, then the value you are looking at is before taking the square root, so it is just the average of the square of the differences.) We can also see that there are 404,710 auction records in this group—that is the total size of our training set. The final piece of information shown here is the decision criterion for the best split that was found, which is to split based on the `coupler_system` column.

一番上のノードは、分割が行われる前の初期モデルで、すべてのデータが 1 つのグループになっているときのものです。これは、最もシンプルなモデルです。これは質問ゼロの結果であり、常にデータセット全体の平均値を予測することになる。この場合、販売価格の対数の値が 10.10 と予測されることがわかります。この場合、平均二乗誤差は 0.48 となります。この平方根は 0.69 である。`(m_rmse`、つまり平均二乗誤差の平方根が表示されない限り、表示される値は平方根を取る前の値であり、差の二乗の平均に過ぎないことを思い出してください) また、このグループには 404,710 件のオークションレコードがあり、これがトレーニングセットの合計サイズであることがわかります。最後に、最適な分割の判断基準として、`coupler_system` の列を基準に分割することを示します。

Moving down and to the left, this node shows us that there were 360,847 auction records for equipment where `coupler_system` was less than 0.5. The average value of our dependent variable in this group is 10.21.

Moving down and to the right from the initial model takes us to the records where coupler_system was greater than 0.5.

このノードでは、coupler_system が 0.5 未満の機器のオークション記録が 360,847 件あることが示されています。このグループにおける従属変数の平均値は、10.21 です。初期モデルから下へ、そして右へ移動すると、coupler_system が 0.5 より大きいレコードにたどり着きます。

The bottom row contains our *leaf nodes*: the nodes with no answers coming out of them, because there are no more questions to be answered. At the far right of this row is the node containing records where coupler_system was greater than 0.5. The average value here is 9.21, so we can see the decision tree algorithm did find a single binary decision that separated high-value from low-value auction results. Asking only about coupler_system predicts an average value of 9.21 versus 10.1

一番下の行には、リーフノードが含まれています。この行の右端には、coupler_system が 0.5 より大きいレコードを含むノードがあります。このノードの平均値は 9.21 であり、決定木のアルゴリズムは、オークションの結果を価値の高いものと低いものに分ける単一の二値決定を発見したことがわかります。coupler_system についてのみ質問すると、平均値は 10.1 に対して 9.21 と予測されます。

Returning back to the top node after the first decision point, we can see that a second binary decision split has been made, based on asking whether YearMade is less than or equal to 1991.5. For the group where this is true (remember, this is now following two binary decisions, based on coupler_system and YearMade) the average value is 9.97, and there are 155,724 auction records in this group. For the group of auctions where this decision is false, the average value is 10.4, and there are 205,123 records. So again, we can see that the decision tree algorithm has successfully split our more expensive auction records into two more groups which differ in value significantly.

最初の決定ポイントの後、トップノードに戻ると、YearMade が 1991.5 以下であるかどうかを尋ねることに基づいて、2 番目のバイナリ決定の分割が行われたことがわかる。これが当てはまるグループ (coupler_system と YearMade に基づく 2 つのバイナリ判定に続いていることを思い出してください) については、平均値は 9.97 で、このグループには 155,724 件のオークションレコードがあります。この判定が偽となったオークションのグループでは、平均値は 10.4 で、205,123 件のレコードが存在する。このように、決定木のアルゴリズムは、より高価なオークションレコードを、価値が大きく異なる 2 つのグループに分けることに成功していることがわかります。

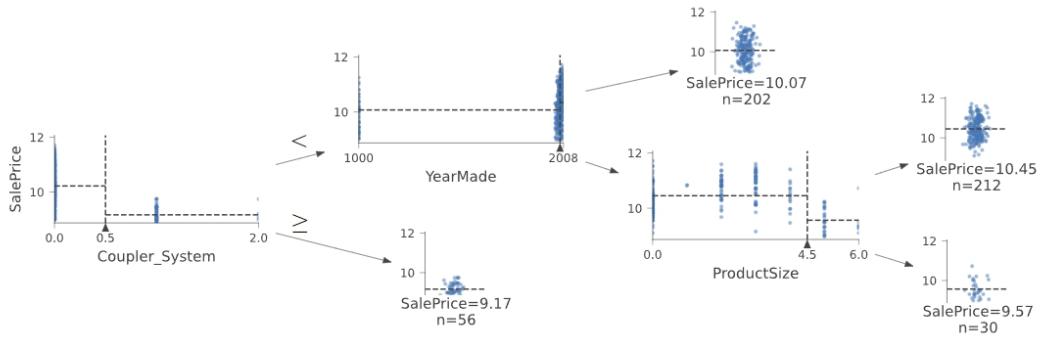
We can show the same information using Terence Parr's powerful `dtreeviz` library:

Terence Parr の強力な `dtreeviz` ライブラリを使って、同じ情報を表示することができます：

In []:

```
samp_idx = np.random.permutation(len(y))[:500]
dtreeviz(m, xs.iloc[samp_idx], y.iloc[samp_idx], xs.columns, dep_var,
          fontname='DejaVu Sans', scale=1.6, label_fontsize=10,
          orientation='LR')
```

Out[]:



This shows a chart of the distribution of the data for each split point. We can clearly see that there's a problem with our YearMade data: there are bulldozers made in the year 1000, apparently! Presumably this is actually just a missing value code (a value that doesn't otherwise appear in the data and that is used as a placeholder in cases where a value is missing). For modeling purposes, 1000 is fine, but as you can see this outlier makes visualization of the values we are interested in more difficult. So, let's replace it with 1950:

これは、各分割点におけるデータの分布を示すグラフです。YearMade のデータに問題があることがよくわかります：1000 年に作られたブルドーザーがあるようです！おそらく、これは単なる欠損値コード（データ中に存在しない値で、値が欠損している場合にプレースホルダーとして使用される）であると思われます。モデリング目的であれば 1000 でも問題ありませんが、ご覧のようにこの異常値は、私たちが関心を持つ値の視覚化を難しくしています。そこで、1950 に置き換えてみましょう：

In []:

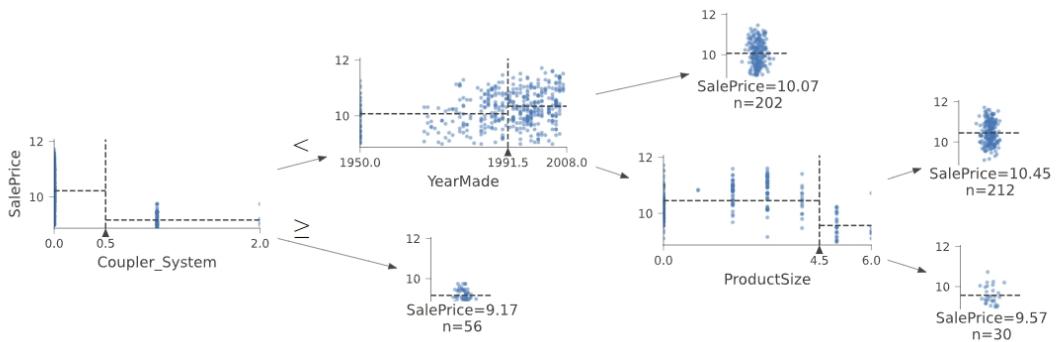
```
xs.loc[xs['YearMade']<1900, 'YearMade'] = 1950
valid_xs.loc[valid_xs['YearMade']<1900, 'YearMade'] = 1950
```

That change makes the split much clearer in the tree visualization, even although it doesn't actually change the result of the model in any significant way. This is a great example of how resilient decision trees are to data issues! この変更により、モデルの結果が大きく変わることはありませんが、ツリーの視覚化で分割がより明確になりました。これは、決定木がデータの問題にいかに強いかを示す良い例です！

In []:

```
m = DecisionTreeRegressor(max_leaf_nodes=4).fit(xs, y)
dtreeviz(m, xs.iloc[samp_idx], y.iloc[samp_idx], xs.columns, dep_var,
         fontname='DejaVu Sans', scale=1.6, label_fontsize=10,
         orientation='LR')
```

Out[]:



Let's now have the decision tree algorithm build a bigger tree. Here, we are not passing in any stopping criteria such as `max_leaf_nodes`:

ここで、決定木アルゴリズムに、より大きな木を作らせることにしましょう。ここでは、`max_leaf_nodes` のような停止基準は渡さない：

In []:

```
m = DecisionTreeRegressor()
m.fit(xs, y);
```

We'll create a little function to check the root mean squared error of our model (`m_rmse`), since that's how the competition was judged:

コンペの判定方法なので、モデルの二乗平均平方根誤差（`m_rmse`）をチェックする小さな関数を作つておくことにします：

In []:

```
def r_mse(pred, y): return round(math.sqrt(((pred - y)**2).mean()), 6)
def m_rmse(m, xs, y): return r_mse(m.predict(xs), y)
```

In []:

```
m_rmse(m, xs, y)
```

Out[]:

0.0

So, our model is perfect, right? Not so fast... remember we really need to check the validation set, to ensure we're not overfitting:

では、このモデルは完璧なのでしょうか？ そうではありません... オーバーフィットしていないことを確認するために、検証セットをチェックする必要があることを思い出してください：

In []:

```
m_rmse(m, valid_xs, valid_y)
```

Out[]:

0.331466

Oops—it looks like we might be overfitting pretty badly. Here's why:

あらら、どうやらかなりオーバーフィッティングしているようです。その理由は以下の通りです：

```
In [ ]:  
m.get_n_leaves(), len(xs)  
Out[ ]:  
(324544, 404710)
```

We've got nearly as many leaf nodes as data points! That seems a little over-enthusiastic. Indeed, sklearn's default settings allow it to continue splitting nodes until there is only one item in each leaf node. Let's change the stopping rule to tell sklearn to ensure every leaf node contains at least 25 auction records:

データポイントとほぼ同じ数のリーフノードがあります！ちょっと気合が入りすぎのような気もします。実際、sklearn のデフォルト設定では、各リーフノードにアイテムが 1つしかなくなるまでノードを分割し続けることができます。停止ルールを変更して、すべてのリーフノードに少なくとも 25 のオークションレコードが含まれていることを確認するように sklearn に指示しよう：

```
In [ ]:  
m = DecisionTreeRegressor(min_samples_leaf=25)  
m.fit(to.train_xs, to.train_y)  
m_rmse(m, xs, y), m_rmse(m, valid_xs, valid_y)  
Out[ ]:  
(0.248562, 0.323396)
```

That looks much better. Let's check the number of leaves again:

だいぶ良くなったようですね。もう一度、葉の数を確認してみましょう：

```
In [ ]:  
m.get_n_leaves()  
Out[ ]:  
12397
```

Much more reasonable!

ずっと合理的です！

A: Here's my intuition for an overfitting decision tree with more leaf nodes than data items. Consider the game Twenty

Questions. In that game, the chooser secretly imagines an object (like, "our television set"), and the guesser gets to pose 20 yes or no questions to try to guess what the object is (like "Is it bigger than a breadbox?"). The guesser is not trying to predict a numerical value, but just to identify a particular object out of the set of all imaginable objects. When your decision tree has more leaves than there are possible objects in your domain, then it is essentially a well-trained guesser. It has learned the sequence of questions needed to identify a particular data item in the training set, and it is "predicting" only by describing that item's value. This is a way of memorizing the training set—i.e., of overfitting.

A: データ項目よりも多くの葉ノードを持つ決定木の過剰適合についての私の直感を紹介します。20の質問」というゲームを考えてみましょう。このゲームでは、選ぶ側が密かにある物体（例えば「私たちのテレビ」）を想像し、当てる側が20のイエスかノーかの質問を投げかけて、その物体が何であるかを推測します（「パン箱より大きいか」のように）。推測者は、数値を予測するのではなく、想像しうるすべての対象物の中から特定の対象物を特定することになります。決定木の葉の数が、その領域で考えられるオブジェクトの数よりも多い場合、その決定木は本質的によく訓練された推測器であると言えます。決定木は、トレーニングセット内の特定のデータ項目を識別するために必要な一連の質問を学習し、その項目の値を記述することによってのみ「予測」しているのです。これは、訓練セットを記憶する方法、つまりオーバーフィッティングです。

Building a decision tree is a good way to create a model of our data. It is very flexible, since it can clearly handle nonlinear relationships and interactions between variables. But we can see there is a fundamental compromise between how well it generalizes (which we can achieve by creating small trees) and how accurate it is on the training set (which we can achieve by using large trees).

決定木の作成は、データのモデルを作成するのに適した方法です。非線形の関係や変数間の相互作用を明確に扱うことができるため、非常に柔軟性があります。しかし、決定木の汎化度（小さな木を作ることで達成できる）と、トレーニングセットでの正確さ（大きな木を使うことで達成できる）の間には、根本的な妥協点があることがわかります。

So how do we get the best of both worlds? We'll show you right after we handle an important missing detail: how to handle categorical variables.

では、どうすれば両者の良いとこ取りができるのでしょうか？ カテゴリー変数の扱い方という、重要な欠落を処理した後にご紹介します。

Categorical Variables

カテゴリカル変数

In the previous chapter, when working with deep learning networks, we dealt with categorical variables by one-hot encoding them and feeding them to an embedding layer. The embedding layer helped the model to discover the meaning of the different levels of these variables (the levels of a categorical variable do not have an intrinsic meaning, unless we manually specify an ordering using Pandas). In a decision tree, we don't have embeddings layers—so how can these untreated categorical variables do anything useful in a decision tree? For instance, how could something like a product code be used?

前章では、ディープラーニングネットワークを扱う際に、カテゴリカル変数をワンホットエンコーディングしてエンベッディング層に供給することで対処しました。エンベッディング層は、モデルが変数の異なるレベルの意味を発見するのに役立ちました（カテゴリカル変数のレベルは、Pandas を使って手動で順序を指定しない限り、固有の意味を持ちません）。決定木では、埋め込み層がないので、未処理のカテゴリカル変数が決定木でどのように役に立つのでしょうか？例えば、商品コードのようなものはどのように利用できるのでしょうか？

The short answer is: it just works! Think about a situation where there is one product code that is far more expensive at auction than any other one. In that case, any binary split will result in that one product code being in some group, and that group will be more expensive than the other group. Therefore, our simple decision tree building algorithm will choose that split. Later during training the algorithm will be able to further split the subgroup that contains the expensive product code, and over time, the tree will home in on that one expensive product.

答えは、「うまくいくから」です！例えば、オークションで他の商品よりはるかに高い商品コードがある場合を考えてみましょう。その場合、2値分割をすると、その1つの商品コードがあるグループに入り、そのグループは他のグループより高価になります。したがって、この単純な決定木構築アルゴリズムは、その分割を選択することになる。その後、学習中に、このアルゴリズムは、高価な製品コードを含むサブグループをさらに分割することができます、時間の経過とともに、ツリーはその高価な1つの製品に近づいていくことになります。

It is also possible to use one-hot encoding to replace a single categorical variable with multiple one-hot-encoded columns, where each column represents a possible level of the variable. Pandas has a `get_dummies` method which does just that.

また、ワンホットエンコーディングを使用して、1つのカテゴリカル変数を複数のワンホットエンコーディングされた列で置き換えることも可能で、各列は変数の可能なレベルを表しています。Pandasには `get_dummies` メソッドがあり、これを実行します。

However, there is not really any evidence that such an approach improves the end result. So, we generally avoid it where possible, because it does end up making your dataset harder to work with. In 2019 this issue was explored in the paper "[Splitting on Categorical Predictors in Random Forests](#)" by Marvin Wright and Inke König, which said:

しかし、このようなアプローチが最終的な結果を向上させるという根拠はありません。そのため、結局はデータセットを扱いにくくすることになるので、可能な限り避けるのが一般的です。2019年、この問題は Marvin Wright と Inke König による論文「Splitting on Categorical Predictors in Random Forests」で検討され、次のように述べられています：

: The standard approach for nominal predictors is to consider all 2^{k-1} partitions of the k predictor categories. However, this exponential relationship produces a large number of potential splits to be evaluated, increasing computational complexity and restricting the possible number of categories in most implementations. For binary classification and regression, it was shown that ordering the predictor categories in each split leads to exactly the same splits as the standard approach. This reduces computational complexity because only $k - 1$ splits have to be considered for a nominal predictor with k categories.

: 名目的な予測因子の標準的なアプローチは、すべての予測因子を考慮することです。

k 個の予測器カテゴリの 2 分割。しかし、この指数関係は、評価されるべき多数の潜在的な分割を生み出し、計算の複雑さを増加させ、ほとんどの実装でカテゴリの可能な数を制限している。2 値分類と回帰については、各分割で予測変数のカテゴリを順番に並べることで、標準アプローチとまったく同じ分割になることが示された。これは、 k 個のカテゴリを持つ名目的な予測変数に対して、 $k - 1$ 個の分割だけを考慮する必要があるので、計算の複雑さを軽減します。

Now that you understand how decisions tree work, it's time for the best-of-both-worlds solution: random forests.

さて、決定木がどのように機能するかを理解したところで、次は両世界のベスト・オブ・ソリューションであるランダム・フォレストを紹介します。

Random Forests

ランダムフォレスト

In 1994 Berkeley professor Leo Breiman, one year after his retirement, published a small technical report called "[Bagging Predictors](#)", which turned out to be one of the most influential ideas in modern machine learning. The report began:

1994 年、バークレー校のレオ・ブレイマン教授は、定年退職の翌年に「Bagging Predictors」という小さな技術レポートを発表したが、これが現代の機械学習において最も影響力のあるアイデアの 1 つであることが判明した。そのレポートはこう始まっている：

: Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. The aggregation averages over the versions... The multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets. Tests... show that bagging can give substantial gains in accuracy. The vital element is the instability of the prediction method. If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.

: 予測変数のバギングとは、予測変数の複数のバージョンを生成し、それらを使って集約された予測変数を得る方法である。集約は、各バージョンを平均化する... 複数のバージョンは、学習セットのブートストラップ・レプリカを作成し、これらを新しい学習セットとして使用することで形成される。テストでは、バギングによって精度が大幅に向上することが示されています。重要なのは、予測手法の不安定さである。学習セットの振動が、構築された予測器に大きな変化をもたらすことができれば、バギングによって精度を向上させることができます。

Here is the procedure that Breiman is proposing:

ブレイマンが提案している手順はこうだ:

1. Randomly choose a subset of the rows of your data (i.e., "bootstrap replicates of your learning set").
データの行の一部をランダムに選択する（すなわち、「学習セットのブートストラップ複製」）。
2. Train a model using this subset.
このサブセットを使ってモデルを訓練する。
3. Save that model, and then return to step 1 a few times.
そのモデルを保存し、ステップ 1 に数回戻る。
4. This will give you a number of trained models. To make a prediction, predict using all of the models, and then take the average of each of those model's predictions.
こうすることで、訓練されたモデルの数が得られます。予測を行うには、すべてのモデルを使って予測し、それらのモデルの各予測値の平均を取ります。

This procedure is known as "bagging." It is based on a deep and important insight: although each of the models trained on a subset of data will make more errors than a model trained on the full dataset, those errors will not be correlated with each other. Different models will make different errors. The average of those errors, therefore, is zero! So if we take the average of all of the models' predictions, then we should end up with a prediction that gets closer and closer to the correct answer, the more models we have. This is an extraordinary result—it means that we can improve the accuracy of nearly any kind of machine learning algorithm by training it multiple times, each time on a different random subset of the data, and averaging its predictions.

この手順は "バギング" として知られています。これは深く重要な洞察に基づいています：データのサブセットで訓練された各モデルは、完全なデータセットで訓練されたモデルよりも多くのエラーを起こしますが、それらのエラーは互いに相関しているわけではありません。モデルによって誤差は異なる。したがって、これらの誤差の平均はゼロとなります！ですから、すべてのモデルの予測値の平均をとると、モデルが増えるほど正解に近づいていく予測値が得られるはずです。つまり、機械学習アルゴリズムの精度を向上させるには、データの異なるランダムな部分集合を複数回学習させ、その予測値を平均化すればよいということです。

In 2001 Leo Breiman went on to demonstrate that this approach to building models, when applied to decision tree building algorithms, was particularly powerful. He went even further than just randomly choosing rows for each model's training, but also randomly selected from a subset of columns when choosing each split in each decision tree. He called this method the *random forest*. Today it is, perhaps, the most widely used and practically important machine learning method.

2001年、レオ・ブレイマンは、このようなモデル構築のアプローチを決定木の構築に適用すると、特に強力であることを実証した。彼は、各モデルの学習で行をランダムに選ぶだけでなく、決定木の各分割を選ぶ際にも、列のサブセットからランダムに選ぶようにした。彼はこの方法を「ランダムフォレスト」と名付けた。現在では、おそらく最も広く使われ、実用上重要な機械学習手法となっている。

In essence a random forest is a model that averages the predictions of a large number of decision trees, which are generated by randomly varying various parameters that specify what data is used to train the tree and other tree parameters. Bagging is a particular approach to "ensembling," or combining the results of multiple models together. To see how it works in practice, let's get started on creating our own random forest!

ランダムフォレストとは、多数の決定木の予測値を平均化したモデルのこと、決定木は、木を訓練するために使用するデータや他の木のパラメータを指定する様々なパラメータをランダムに変化させることで生成されます。バギングとは、「アンサンブル」、つまり複数のモデルの結果を一緒に組み合わせるための特別なアプローチです。実際にどのように機能するかを確認するために、独自のランダムフォレストの作成に取りかかりましょう！

In []:

```
#hide  
# pip install --pre -f https://sklearn-nightly.scdn8.secure.raxcdn.com scikit-learn -U
```

Creating a Random Forest

ランダムフォレストを作成する

We can create a random forest just like we created a decision tree, except now, we are also specifying parameters that indicate how many trees should be in the forest, how we should subset the data items (the rows), and how we should subset the fields (the columns).

決定木を作成するのと同じようにランダムフォレストを作成することができますが、ここでは、フォレスト内に何本の木を配置するか、データ項目（行）をどのようにサブセット

するか、フィールド（列）をどのようにサブセットするかを示すパラメータも指定しています。

In the following function definition `n_estimators` defines the number of trees we want, `max_samples` defines how many rows to sample for training each tree, and `max_features` defines how many columns to sample at each split point (where 0.5 means "take half the total number of columns"). We can also specify when to stop splitting the tree nodes, effectively limiting the depth of the tree, by including the same `min_samples_leaf` parameter we used in the last section. Finally, we pass `n_jobs=-1` to tell `sklearn` to use all our CPUs to build the trees in parallel. By creating a little function for this, we can more quickly try different variations in the rest of this chapter:

以下の関数定義では、`n_estimators` は必要な木の数を、`max_samples` は各木をトレーニングするためにサンプリングする行数を、`max_features` は各分割点でサンプリングする列数を定義します（ここで 0.5 は「全列数の半分を取る」ことを意味しています）。また、前節で使用した `min_samples_leaf` パラメータを指定することで、木のノードの分割を停止するタイミングを指定し、木の深さを効果的に制限することができます。最後に、`n_jobs=-1` を渡して、`sklearn` に全 CPU を使用して並列に木を構築するように指示します。このために小さな関数を作成することで、この章の残りの部分でより迅速に様々なバリエーションを試すことができます：

In []:

```
def rf(xs, y, n_estimators=40,
       max_samples=200_000, max_features=0.5, min_samples_leaf=5, **kwargs):
    return RandomForestRegressor(n_jobs=-1, n_estimators=n_estimators,
                                 max_samples=max_samples, max_features=max_features,
                                 min_samples_leaf=min_samples_leaf, oob_score=True).fit(xs, y)
```

In []:

```
m = rf(xs, y);
```

Our validation RMSE is now much improved over our last result produced by the `DecisionTreeRegressor`, which made just one tree using all the available data:

検証の RMSE は、`DecisionTreeRegressor` によって生成された最後の結果よりもはるかに改善されています。この結果、利用可能なすべてのデータを使用して 1 つのツリーだけを作成しました：

In []:

```
m_rmse(m, xs, y), m_rmse(m, valid_xs, valid_y)
```

Out[]:

```
(0.170917, 0.233975)
```

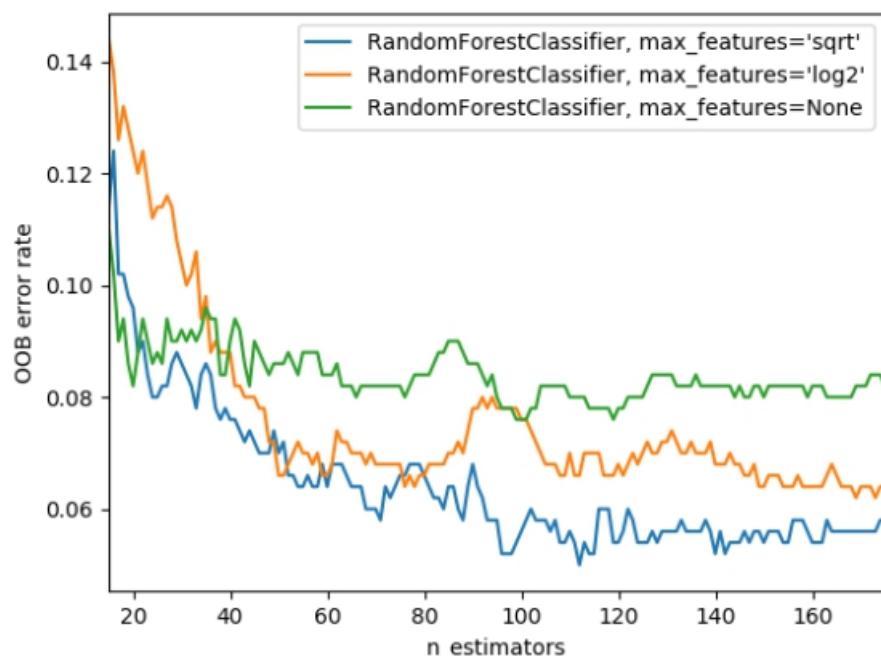
One of the most important properties of random forests is that they aren't very sensitive to the hyperparameter choices, such as `max_features`. You can set `n_estimators` to as high a number as you have time to train—the more trees you have, the more accurate the model will be. `max_samples` can often be left at its default, unless you have

over 200,000 data points, in which case setting it to 200,000 will make it train faster with little impact on accuracy. `max_features=0.5` and `min_samples_leaf=4` both tend to work well, although sklearn's defaults work well too.

ランダムフォレストの最も重要な特性の1つは、`max_features`などのハイパーパラメータの選択にあまり影響されないということです。`max_features=0.5`と`min_samples_leaf=4`はどちらもうまくいく傾向がありますが、sklearnのデフォルトもうまくいくでしょう。

The sklearn docs [show an example](#) of the effects of different `max_features` choices, with increasing numbers of trees. In the plot, the blue plot line uses the fewest features and the green line uses the most (it uses all the features). As you can see in <<`max_features`>>, the models with the lowest error result from using a subset of features but with a larger number of trees.

sklearnのドキュメントには、木の本数を増やしながら`max_features`の選択を変えた場合の効果の例が示されています。プロットでは、青いプロット線が最も少ない特徴量を使い、緑の線が最も多く使っています（すべての特徴量を使用しています）。<>でわかるように、最も誤差の少ないモデルは、特徴のサブセットを使いながら、より多くの木を使った結果である。



To see the impact of `n_estimators`, let's get the predictions from each individual tree in our forest (these are in the `estimators_` attribute):

`n_estimators`の影響を見るために、森の中の個々の木から予測値を取得してみましょう（これらは `estimators_` 属性にあります）：

In []:

```
preds = np.stack([t.predict(valid_xs) for t in m.estimators_])
```

As you can see, `preds.mean(0)` gives the same results as our random forest:

見ての通り、`preds.mean(0)`はランダムフォレストと同じ結果を出しています：

```
r_mse(preds.mean(0), valid_y)
```

In []:

0.233975

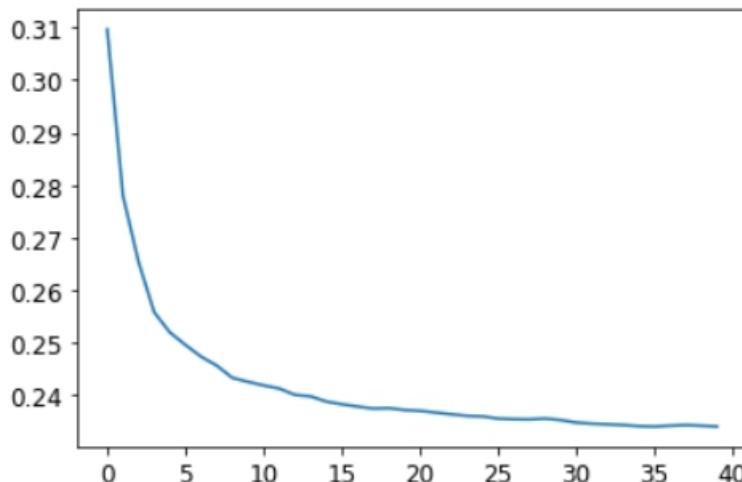
Out[]:

Let's see what happens to the RMSE as we add more and more trees. As you can see, the improvement levels off quite a bit after around 30 trees:

さらに木を増やしていくと、RMSE はどうなるか見てみましょう。見ての通り、30 本程度でかなり改善されていることがわかります：

```
plt.plot([r_mse(preds[i+1].mean(0), valid_y) for i in range(40)]);
```

In []:



The performance on our validation set is worse than on our training set. But is that because we're overfitting, or because the validation set covers a different time period, or a bit of both? With the existing information we've seen, we can't tell. However, random forests have a very clever trick called *out-of-bag* (OOB) error that can help us with this (and more!).

検証セットでの性能はトレーニングセットよりも悪くなっています。しかし、これはオーバーフィッティングのせいなのか、それとも検証セットが異なる期間をカバーしているせいなのか、あるいはその両方なのでしょうか？これまで見てきた既存の情報ではわかりません。しかし、ランダムフォレストには、袋外不出（OOB）エラーという非常に巧妙なトリックがあり、これを利用することができます（他にもあります！）。

Out-of-Bag Error

アウトオブバッグエラー

Recall that in a random forest, each tree is trained on a different subset of the training data. The OOB error is a way of measuring prediction error on the training set by only including in the calculation of a row's error trees where that row was *not* included in training. This allows us to see whether the model is overfitting, without needing a separate validation set.

ランダムフォレストでは、各木は訓練データの異なるサブセットで訓練されることを思い出してください。OOB エラーは、トレーニングセットの予測誤差を測定する方法で、ある行の誤差の計算に、その行がトレーニングに含まれていない木のみを含める。これによって、別途検証セットを用意しなくても、モデルがオーバーフィットしているかどうかを確認することができます。

A: My intuition for this is that, since every tree was trained with a different randomly selected subset of rows, out-of-bag error is a little like imagining that every tree therefore also has its own validation set. That validation set is simply the rows that were not selected for that tree's training.

A: 私の直感では、すべてのツリーはランダムに選択された異なる行のサブセットでトレーニングされているので、袋小路誤差は、すべてのツリーが独自の検証セットを持っていると想像するのと少し似ています。この検証セットは、そのツリーのトレーニングで選択されなかった行のことです。

This is particularly beneficial in cases where we have only a small amount of training data, as it allows us to see whether our model generalizes without removing items to create a validation set. The OOB predictions are available in the `oob_prediction_` attribute. Note that we compare them to the training labels, since this is being calculated on trees using the training set.

これは、トレーニングデータが少ない場合に特に有効で、検証セットを作成するために項目を削除することなく、モデルが一般化するかどうかを確認することができます。OOB 予測は、`oob_prediction_` 属性で利用できます。これはトレーニングセットを使ったツリーで計算されるため、トレーニングラベルと比較することに注意してください。

In []:

```
r_mse(m.oob_prediction_, y)
```

Out[]:

0.210681

We can see that our OOB error is much lower than our validation set error. This means that something else is causing that error, in *addition* to normal generalization error. We'll discuss the reasons for this later in this chapter.

OOB の誤差は、検証セットの誤差よりもずっと小さいことがわかります。これは、通常の汎化誤差に加えて、何か他のものがその誤差の原因になっていることを意味します。この理由については、この章の後半で説明します。

This is one way to interpret our model's predictions—let's focus on more of those now.

これは、モデルの予測値を解釈する一つの方法です。

Model Interpretation

モデルの解釈

For tabular data, model interpretation is particularly important. For a given model, the things we are most likely to be interested in are:

表形式データの場合、モデルの解釈は特に重要である。あるモデルについて、私たちが最も関心を持ちやすいのは次のようなことです：

- How confident are we in our predictions using a particular row of data?
特定のデータ行を使った予測にどの程度の自信があるか？
- For predicting with a particular row of data, what were the most important factors, and how did they influence that prediction?
特定の行のデータを使って予測する場合、最も重要な要因は何か、そしてそれらがその予測にどのような影響を与えたか？
- Which columns are the strongest predictors, which can we ignore?
どの列が最も強い予測因子で、どの列は無視できるのか？
- Which columns are effectively redundant with each other, for purposes of prediction?
予測の目的上、どの列が互いに効果的に冗長なのか？
- How do predictions vary, as we vary these columns?
これらの列を変化させると、予測値はどのように変化するのか？

As we will see, random forests are particularly well suited to answering these questions. Let's start with the first one!

これから説明するように、ランダムフォレストはこれらの質問に答えるのに特に適している。まず、最初の質問から始めましょう！

Tree Variance for Prediction Confidence

予測の信頼度に対する木の変化

We saw how the model averages the individual tree's predictions to get an overall prediction—that is, an estimate of the value. But how can we know the confidence of the estimate? One simple way is to use the standard deviation of predictions across the trees, instead of just the mean. This tells us the *relative* confidence of predictions. In general, we would want to be more cautious of using the results for rows where trees give very different results (higher standard deviations), compared to cases where they are more consistent (lower standard deviations).

このモデルでは、個々の木の予測値を平均して全体的な予測値、つまり値の推定値を得ることを確認しました。しかし、その推定値の信頼度を知るにはどうすればよいのでしょうか。一つの簡単な方法は、平均値だけでなく、樹木全体の予測値の標準偏差を使うことで

す。これにより、予測値の相対的な信頼度がわかります。一般に、樹木の結果が大きく異なる（標準偏差が大きい）行の結果を、より一貫している（標準偏差が小さい）ケースと比較して、より慎重に使用したいものである。

In the earlier section on creating a random forest, we saw how to get predictions over the validation set, using a Python list comprehension to do this for each tree in the forest:

ランダムフォレストを作成する前のセクションで、検証セットの予測値を取得する方法を見ました。Python のリスト内包を使用して、フォレストの各木に対してこれを行います：

```
In [ ]:  
preds = np.stack([t.predict(valid_xs) for t in m.estimators_])  
In [ ]:  
preds.shape  
Out[ ]:  
(40, 7988)
```

Now we have a prediction for every tree and every auction (40 trees and 7,988 auctions) in the validation set.

これで、検証セットのすべての木とすべてのオークション（40 本の木と 7,988 のオークション）に対する予測値ができました。

Using this we can get the standard deviation of the predictions over all the trees, for each auction:

これを用いて、すべての木と各オークションの予測値の標準偏差を得ることができます：

```
In [ ]:  
preds_std = preds.std(0)
```

Here are the standard deviations for the predictions for the first five auctions—that is, the first five rows of the validation set:

最初の 5 つのオークション、つまり検証セットの最初の 5 行の予測値の標準偏差は以下の通りである：

```
In [ ]:  
preds_std[:5]  
Out[ ]:  
array([0.25065395, 0.11043862, 0.08242067, 0.26988508, 0.15730173])
```

As you can see, the confidence in the predictions varies widely. For some auctions, there is a low standard deviation because the trees agree. For others it's higher, as the trees don't agree. This is information that would be useful in a production setting; for instance, if you were using this model to decide what items to bid on at auction, a low-confidence prediction might cause you to look more carefully at an item before you made a bid.

ご覧のように、予測の信頼度は大きく変化します。あるオーケションでは、ツリーが一致するため標準偏差が低くなっています。あるオーケションでは、木が一致するため標準偏差が低く、あるオーケションでは、木が一致しないため標準偏差が高くなるのである。例えば、このモデルを使ってオーケションで落札する商品を決める場合、信頼度の低い予測があれば、落札する前にその商品をもっと注意深く見るようになるかもしれませんね。

Feature Importance

機能の重要性

It's not normally enough just to know that a model can make accurate predictions—we also want to know *how* it's making predictions. *feature importance* gives us insight into this. We can get these directly from sklearn's random forest by looking in the `feature_importances_` attribute. Here's a simple function we can use to pop them into a DataFrame and sort them:

通常、モデルが正確な予測を行うことができるというだけでは不十分で、どのように予測を行っているのかも知りたいものです。`feature_importances_`属性を見ることで、sklearnのランダムフォレストから直接取得することができます。これを DataFrame に取り込み、ソートするための簡単な関数を紹介します：

In []:

```
def rf_feat_importance(m, df):
    return pd.DataFrame({'cols':df.columns, 'imp':m.feature_importances_}
    ).sort_values('imp', ascending=False)
```

The feature importances for our model show that the first few most important columns have much higher importance scores than the rest, with (not surprisingly) `YearMade` and `ProductSize` being at the top of the list:

このモデルの特徴インポータンスを見ると、最初の数列の重要度が他の列よりもはるかに高く、（驚くことではないが）`YearMade` と `ProductSize` がトップであることがわかる：

In []:

```
fi = rf_feat_importance(m, xs) fi[:10]
```

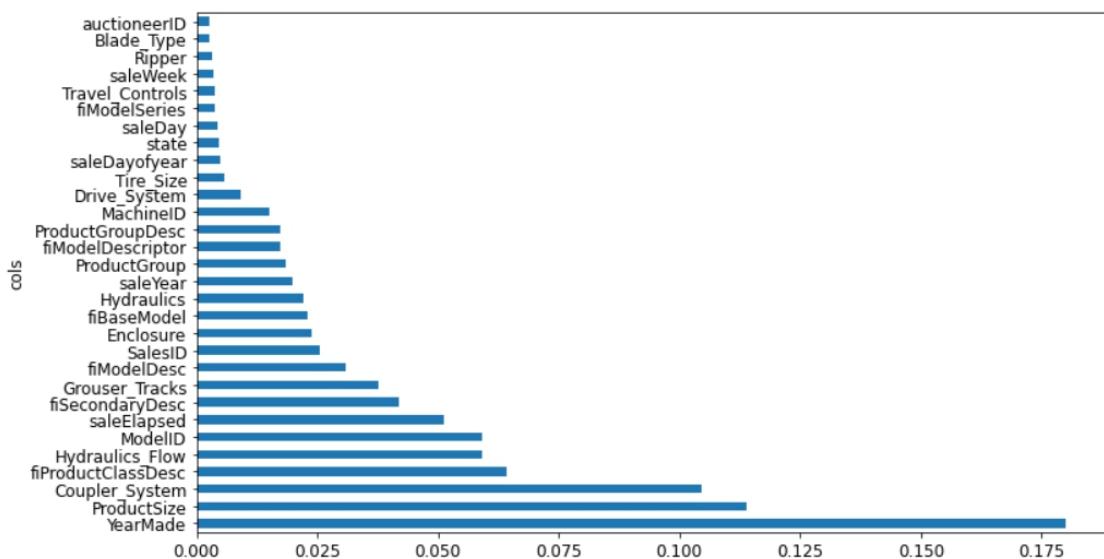
	cols	imp
59	YearMade	0.180070
7	ProductSize	0.113915
31	Coupler_System	0.104699
8	fiProductClassDesc	0.064118
33	Hydraulics_Flow	0.059110
56	ModelID	0.059087
51	saleElapsed	0.051231
4	fiSecondaryDesc	0.041778
32	Grouser_Tracks	0.037560
2	fiModelDesc	0.030933

A plot of the feature importances shows the relative importances more clearly:

特徴のインポートをプロットすると、相対的なインポートをより明確に示すことができる:

In []:

```
def plot_fi(fi):
    return fi.plot('cols', 'imp', 'barh', figsize=(12,7), legend=False)
plot_fi(fi[:30]);
```



The way these importances are calculated is quite simple yet elegant. The feature importance algorithm loops through each tree, and then recursively explores each branch. At each branch, it looks to see what feature was used for that split, and how much the model improves as a result of that split. The improvement (weighted by the number of rows in that group) is added to the importance score for that feature. This is summed across all branches of all trees, and finally the scores are normalized such that they add to 1.

これらの重要度の計算方法は、非常にシンプルでありながらエレガントです。特徴量重要度アルゴリズムは、各ツリーをループし、各分岐を再帰的に探索します。各分岐では、その分岐にどのような特徴が使われたのか、また、その分岐の結果、モデルがどれだけ改善されたのかを確認します。その改善度（そのグループの行数で重み付け）は、その特徴の重要度スコアに加算されます。これをすべての木のすべての枝で合計し、最後にスコアが 1 になるように正規化する。

Removing Low-Importance Variables

重要度の低い変数の削除

It seems likely that we could use just a subset of the columns by removing the variables of low importance and still get good results. Let's try just keeping those with a feature importance greater than 0.005:

重要度の低い変数を削除することで、列のサブセットだけを使用しても、良い結果が得られる可能性があるようです。ここでは、特徴量の重要度が 0.005 より大きいものだけを残すようにしてみます：

```
In [ ]:  
to_keep = fi[fi.imp>0.005].cols len(to_keep)
```

```
Out[ ]:
```

```
21
```

We can retrain our model using just this subset of the columns:

このカラムのサブセットだけを使ってモデルを再トレーニングすることができます：

```
In [ ]:  
xs_imp = xs[to_keep]  
valid_xs_imp = valid_xs[to_keep]
```

```
In [ ]:
```

```
m = rf(xs_imp, y)
```

And here's the result:

そして、その結果がこちらです：

```
In [ ]:  
m_rmse(m, xs_imp, y), m_rmse(m, valid_xs_imp, valid_y)
```

```
Out[ ]:
```

```
(0.181204, 0.230329)
```

Our accuracy is about the same, but we have far fewer columns to study:

精度はほぼ同じですが、調査する列の数が圧倒的に少なくなっています：

In []:

```
len(xs.columns), len(xs_imp.columns)
```

Out[]:

```
(66, 21)
```

We've found that generally the first step to improving a model is simplifying it—78 columns was too many for us to study them all in depth! Furthermore, in practice often a simpler, more interpretable model is easier to roll out and maintain.

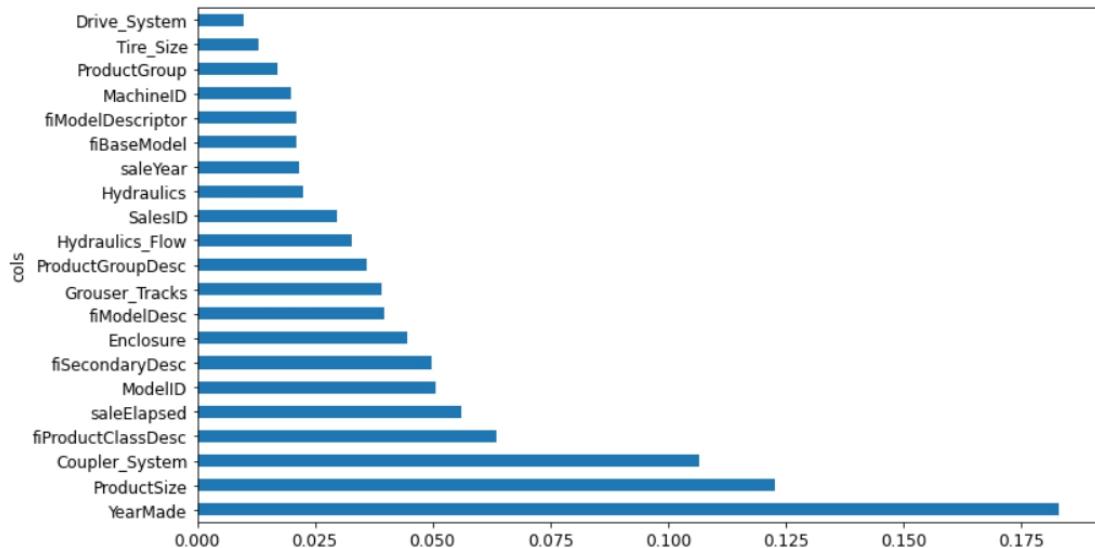
私たちは、モデルを改善するための最初のステップは、一般的にそれを単純化することであることを発見しました-78 の列はあまりにも多く、すべてを深く研究することはできませんでした！さらに、実際には、よりシンプルで解釈しやすいモデルの方が、展開や維持が容易な場合が多いのです。

This also makes our feature importance plot easier to interpret. Let's look at it again:

また、機能重要度プロットも解釈しやすくなります。もう一度見てみましょう：

In []:

```
plot_fi(rf_feat_importance(m, xs_imp));
```



One thing that makes this harder to interpret is that there seem to be some variables with very similar meanings: for example, ProductGroup and ProductGroupDesc. Let's try to remove any redundant features.

例えば、ProductGroup と ProductGroupDesc のように、非常に似た意味を持つ変数があることが、このプロットを難しくしています。冗長な機能を削除してみましょう。

Removing Redundant Features

冗長化された機能を削除する

Let's start with:

で始めましょう：

In []:

```
cluster_columns(xs_imp)
```

In this chart, the pairs of columns that are most similar are the ones that were merged together early, far from the "root" of the tree at the left. Unsurprisingly, the fields ProductGroup and ProductGroupDesc were merged quite early, as were saleYear and saleElapsed and fiModelDesc and fiBaseModel. These might be so closely correlated they are practically synonyms for each other.

この図では、最も似ている列のペアは、左のツリーの「ルート」から遠い、早い時期に一緒にマージされたものです。当然のことながら、ProductGroup と ProductGroupDesc というフィールドはかなり早い段階でマージされましたし、saleYear と saleElapsed や fiModelDesc と fiBaseModel もそうでした。これらは非常に密接な相関があり、実質的にお互いの同義語と言えるかもしれません。

note: Determining Similarity: The most similar pairs are found by calculating the *rank correlation*, which means that all the values are replaced with their *rank* (i.e., first, second, third, etc. within the column), and then the *correlation* is calculated. (Feel free to skip over this minor detail though, since it's not going to come up again in the book!)

note: 類似性の判断: つまり、すべての値をランク（列内の 1 位、2 位、3 位など）に置き換えてから、相関を計算するのです。（この些細なことは、この本では二度と出てきませんので、自由に読み飛ばしてください！）

Let's try removing some of these closely related features to see if the model can be simplified without impacting the accuracy. First, we create a function that quickly trains a random forest and returns the OOB score, by using a lower `max_samples` and higher `min_samples_leaf`. The OOB score is a number returned by sklearn that ranges between 1.0 for a perfect model and 0.0 for a random model. (In statistics it's called R^2 , although the details aren't important for this explanation.) We don't need it to be very accurate—we're just going to use it to compare different models, based on removing some of the possibly redundant columns:

精度に影響を与えることなくモデルを簡略化できるかどうか、これらの密接に関連する特徴をいくつか取り除いてみましょう。まず、`max_samples` を低く、`min_samples_leaf` を高くして、ランダムフォレストを素早く学習させ、OOB スコアを返す関数を作成します。OOB スコアは sklearn が返す数値で、完璧なモデルなら 1.0、ランダムなモデルなら 0.0 の範囲です。（統計学では R^2 と呼ばれます、この説明では詳細は重要ではありません）。冗長と思われる列を削除した上で、異なるモデルを比較するために使用するだけです：

In []:

```
def get_oob(df):
```

```

m = RandomForestRegressor(n_estimators=40, min_samples_leaf=15,
                         max_samples=50000, max_features=0.5, n_jobs=-1, oob_score=True)
m.fit(df, y)
return m.oob_score_

```

Here's our baseline:

これがベースラインです:

In []:
get_oob(xs_imp)
Out[]:
0.8768243241012634

Now we try removing each of our potentially redundant variables, one at a time:

ここで、冗長となりうる変数を 1 つずつ削除してみます:

In []:
{c:get_oob(xs_imp.drop(c, axis=1)) for c in (
'saleYear', 'saleElapsed', 'ProductGroupDesc', 'ProductGroup',
'fiModelDesc', 'fiBaseModel',
'Hydraulics_Flow', 'Grouser_Tracks', 'Coupler_System')}
Out[]:
{'saleYear': 0.8766429216799364,
'saleElapsed': 0.8725120463477113,
'ProductGroupDesc': 0.8773289113713139,
'ProductGroup': 0.8768277447901079,
'fiModelDesc': 0.8760365396140016,
'fiBaseModel': 0.8769194097714894,
'Hydraulics_Flow': 0.8775975083138958,
'Grouser_Tracks': 0.8780246481379101,
'Coupler_System': 0.8780158691125818}

Now let's try dropping multiple variables. We'll drop one from each of the tightly aligned pairs we noticed earlier.

Let's see what that does:

今度は、複数の変数をドロップしてみましょう。先ほど気づいた、ぴったりと並んだペアからそれぞれ 1 つずつ落としてみます。その結果、どうなるか見てみましょう:

In []:
to_drop = ['saleYear', 'ProductGroupDesc', 'fiBaseModel', 'Grouser_Tracks']

```
get_oob(xs_imp.drop(to_drop, axis=1))
```

Out[]:

```
0.8747772191306009
```

Looking good! This is really not much worse than the model with all the fields. Let's create DataFrames without these columns, and save them:

見えはいい！これは、すべてのフィールドを持つモデルよりも、はるかに悪いものではありません。では、これらのカラムを含まない DataFrame を作成し、保存してみましょう：

In []:

```
xs_final = xs_imp.drop(to_drop, axis=1)
valid_xs_final = valid_xs_imp.drop(to_drop, axis=1)
```

In []:

```
save_pickle(path/'xs_final.pkl', xs_final)
save_pickle(path/'valid_xs_final.pkl', valid_xs_final)
```

We can load them back later with:

で後でロードし直せばいい：

In []:

```
xs_final = load_pickle(path/'xs_final.pkl')
valid_xs_final = load_pickle(path/'valid_xs_final.pkl')
```

Now we can check our RMSE again, to confirm that the accuracy hasn't substantially changed.

ここでもう一度 RMSE を確認し、精度が大きく変化していないことを確認します。

In []:

```
m = rf(xs_final, y)
m_rmse(m, xs_final, y), m_rmse(m, valid_xs_final, valid_y)
```

Out[]:

```
(0.183426, 0.231894)
```

By focusing on the most important variables, and removing some redundant ones, we've greatly simplified our model. Now, let's see how those variables affect our predictions using partial dependence plots.

最も重要な変数に焦点を当て、冗長な変数を削除することで、モデルを大幅に簡略化することができました。では、それらの変数が予測にどのような影響を与えるか、部分依存性プロットを使って見てみましょう。

Partial Dependence

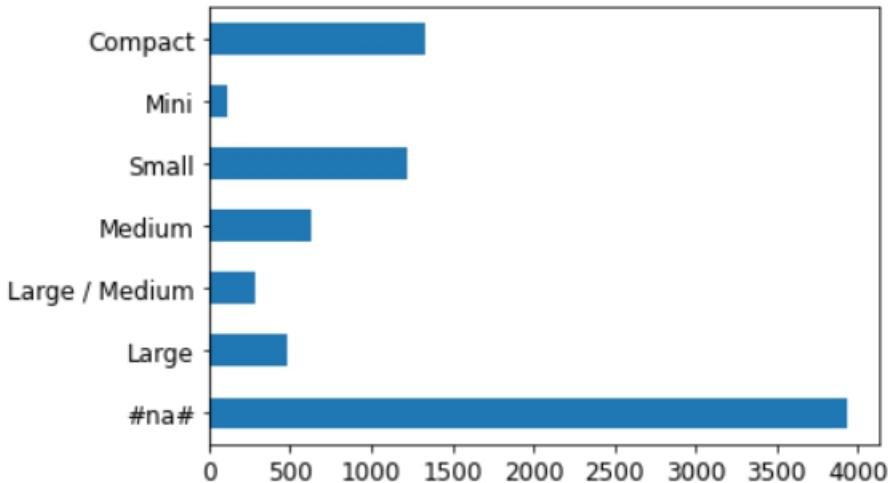
部分的な依存関係

As we've seen, the two most important predictors are ProductSize and YearMade. We'd like to understand the relationship between these predictors and sale price. It's a good idea to first check the count of values per category (provided by the Pandas value_counts method), to see how common each category is:

これまで見てきたように、最も重要な予測因子は ProductSize と YearMade の 2 つです。これらの予測因子と販売価格との関係を理解したいと思います。まず、カテゴリごとの値のカウント（Pandas の value_counts メソッドで提供される）を確認し、各カテゴリがどれだけ一般的であるかを確認するのは良いアイデアです：

In []:

```
p = valid_xs_final[ProductSize].value_counts(sort=False).plot.barh()
c = to.classes[ProductSize]
plt.yticks(range(len(c)), c);
```



The largest group is #na#, which is the label fastai applies to missing values.

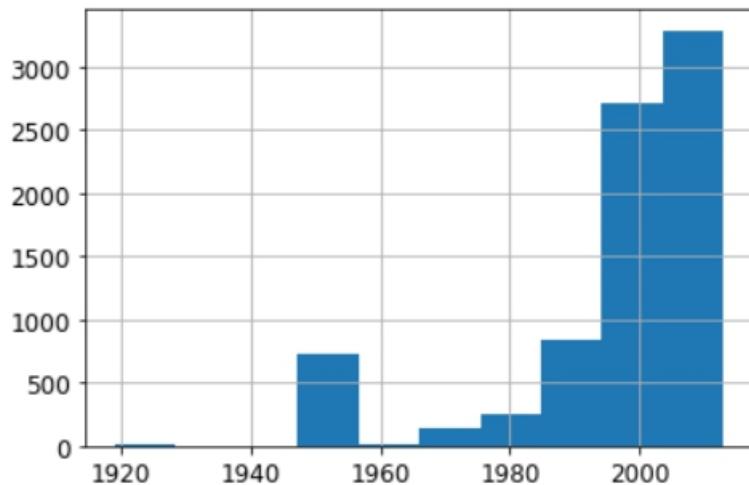
最も大きなグループは#na#で、これは fastai が欠損値に適用するラベルである。

Let's do the same thing for YearMade. Since this is a numeric feature, we'll need to draw a histogram, which groups the year values into a few discrete bins:

同じことを YearMade についてもやってみましょう。これは数値の特徴なので、ヒストグラムを描く必要があり、年の値をいくつかの離散的な bin にグループ化します：

In []:

```
ax = valid_xs_final[YearMade].hist()
```



Other than the special value 1950 which we used for coding missing year values, most of the data is from after 1990.

欠損年のコーディングに使用した特別な値 1950 以外は、ほとんどのデータが 1990 年以降のものである。

Now we're ready to look at *partial dependence plots*. Partial dependence plots try to answer the question: if a row varied on nothing other than the feature in question, how would it impact the dependent variable?

さて、部分依存性プロットを見る準備が整いました。部分依存度プロットは、「もある行が、問題の特徴以外の要素で変化したら、従属変数にどのような影響を与えるか」という質問に答えようとします。

For instance, how does YearMade impact sale price, all other things being equal?

例えば、YearMade が販売価格にどのような影響を与えるのか、他の条件がすべて同じであるのか。

To answer this question, we can't just take the average sale price for each YearMade. The problem with that approach is that many other things vary from year to year as well, such as which products are sold, how many products have air-conditioning, inflation, and so forth. So, merely averaging over all the auctions that have the same YearMade would also capture the effect of how every other field also changed along with YearMade and how that overall change affected price.

この問い合わせるには、各 YearMade の平均販売価格を取るだけではダメなのです。その方法の問題点は、どの製品が売れたか、どれだけの製品にエアコンがついたか、インフレ率など、他の多くの事柄も年ごとに変化することです。そのため、同じ YearMade を持つすべてのオーバークションを平均化するだけでは、YearMade と一緒に他のあらゆる分野も変化し、その全体的な変化が価格にどのような影響を与えたかという影響をも捉えてしまうことになる。

Instead, what we do is replace every single value in the YearMade column with 1950, and then calculate the predicted sale price for every auction, and take the average over all auctions. Then we do the same for 1951, 1952, and so forth until our final year of 2011. This isolates the effect of only YearMade (even if it does so by averaging over some imagined records where we assign a YearMade value that might never actually exist alongside some other values).

そこで、YearMade の欄の値をすべて 1950 に置き換えて、すべてのオークションの予測落札価格を計算し、全オークションの平均値をとります。次に、1951 年、1952 年……と、最終年である 2011 年まで同じことをするのである。これにより、YearMade の効果のみを分離することができます（たとえ、実際には存在しないかもしれない YearMade の値を他の値と一緒に割り当てた想像上の記録を平均化することによってそうしているのだとしても）。

A: If you are philosophically minded it is somewhat dizzying to contemplate the different kinds of hypotheticality that we are juggling to make this calculation. First, there's the fact that *every* prediction is hypothetical, because we are not noting empirical data. Second, there's the point that we're *not* merely interested in asking how sale price would change if we changed YearMade and everything else along with it. Rather, we're very specifically asking, how sale price would change in a hypothetical world where only YearMade changed. Phew! It is impressive that we can ask such questions. I recommend Judea Pearl and Dana Mackenzie's recent book on causality, *The Book of Why* (Basic Books), if you're interested in more deeply exploring formalisms for analyzing these subtleties.

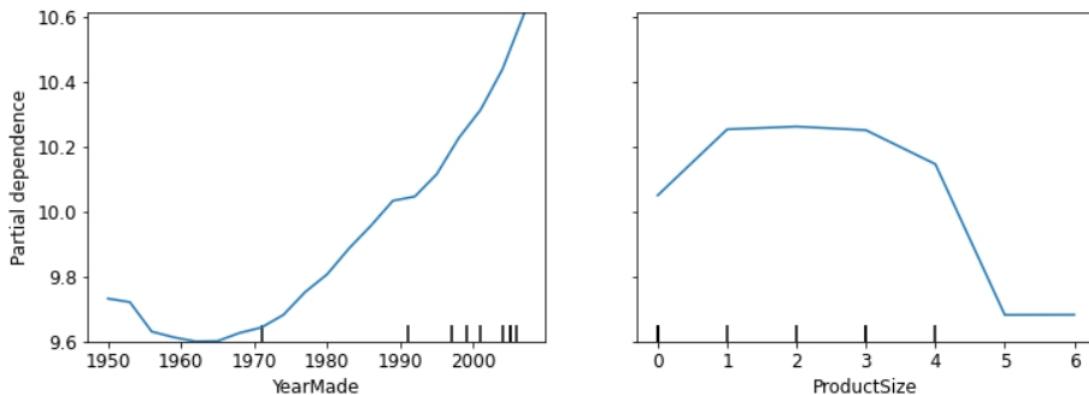
A: もしあなたが哲学的な思考をお持ちなら、この計算をするために私たちが扱っているさまざまな種類の仮説について考えるのは、少々めまいがすることです。まず、私たちは経験的なデータに注目しているわけではないので、すべての予測が仮説的であるという事実があります。第二に、私たちは、単に YearMade とそれに付随する他のすべてを変えたら販売価格がどのように変化するかということに興味があるわけではないという点です。むしろ、YearMade だけが変化した仮想の世界では、販売価格はどう変化するのか、ということを具体的に聞いています。ふうー！ そんな質問ができるなんて、すごいですね。このような微妙な問題を分析するための形式論に興味がある方は、Judea Pearl と Dana Mackenzie の因果関係に関する最近の本、*The Book of Why* (Basic Books)をお薦めします。

With these averages, we can then plot each of these years on the x-axis, and each of the predictions on the y-axis. This, finally, is a partial dependence plot. Let's take a look:

この平均値を用いて、X 軸に各年代を、Y 軸に各予測値をプロットすることができます。これが、部分依存のプロットです。見てみましょう：

In []:

```
from sklearn.inspection import plot_partial_dependence
fig,ax = plt.subplots(figsize=(12, 4))
plot_partial_dependence(m, valid_xs_final, ['YearMade','ProductSize'],
grid_resolution=20, ax=ax);
```



Looking first of all at the YearMade plot, and specifically at the section covering the years after 1990 (since as we noted this is where we have the most data), we can see a nearly linear relationship between year and price. Remember that our dependent variable is after taking the logarithm, so this means that in practice there is an exponential increase in price. This is what we would expect: depreciation is generally recognized as being a multiplicative factor over time, so, for a given sale date, varying year made ought to show an exponential relationship with sale price.

まず YearMade のプロットを見ると、特に 1990 年以降のセクションを見ると（前述の通り、ここが最もデータが豊富な場所だから）、年と価格の間にほぼ直線的な関係があることがわかります。従属変数が対数をとった後であることを思い出してください。つまり、実際には価格は指数関数的に上昇します。これは私たちが期待することです。減価償却は一般に時間の経過とともに乗算的な要因として認識されるので、ある販売日について、製造年を変化させると、販売価格と指数関数的な関係を示すはずです。

The ProductSize partial plot is a bit concerning. It shows that the final group, which we saw is for missing values, has the lowest price. To use this insight in practice, we would want to find out *why* it's missing so often, and what that *means*. Missing values can sometimes be useful predictors—it entirely depends on what causes them to be missing. Sometimes, however, they can indicate *data leakage*.

ProductSize の部分プロットは、少し気になるところです。これは、最終グループ（欠損値であることがわかった）が最も低い価格を持っていることを示しています。この洞察を実際に使うには、なぜ欠落が多いのか、そしてそれが何を意味するのかを調べたいところです。欠損値は、時に有用な予測因子となり得ますが、それは欠損の原因に完全に依存します。しかし、時にはデータの漏洩を示すこともあります。

Data Leakage

データ漏えい

In the paper "[Leakage in Data Mining: Formulation, Detection, and Avoidance](#)", Shachar Kaufman, Saharon Rosset, and Claudia Perlich describe leakage as:

論文「データマイニングにおけるリーケージ（Leakage in Data Mining: Formulation, Detection, and Avoidance）」という論文で、Shachar Kaufman, Saharon Rosset, Claudia Perlich は、リーケージとは次のように説明している：

: The introduction of information about the target of a data mining problem, which should not be legitimately available to mine from. A trivial example of leakage would be a model that uses the target itself as an input, thus concluding for example that 'it rains on rainy days'. In practice, the introduction of this illegitimate information is unintentional, and facilitated by the data collection, aggregation and preparation process.

：データマイニング問題の対象について、そこからマイニングするために正当に利用可能であってはならない情報を導入すること。リーケージの些細な例としては、ターゲットそのものを入力として使用し、例えば「雨の日には雨が降る」と結論づけるようなモデルがあげられる。実際には、このような非合法な情報の導入は意図的ではなく、データの収集、集計、準備のプロセスによって促進される。

They give as an example:

その一例として、次のようなものがある：

: A real-life business intelligence project at IBM where potential customers for certain products were identified, among other things, based on keywords found on their websites. This turned out to be leakage since the website content used for training had been sampled at the point in time where the potential customer has already become a customer, and where the website contained traces of the IBM products purchased, such as the word 'Websphere' (e.g., in a press release about the purchase or a specific product feature the client uses).

: IBM で実際に行われたビジネス・インテリジェンス・プロジェクトでは、ある製品の潜在的な顧客を、そのウェブサイトにあるキーワードに基づいて特定することなどが行われた。これは、トレーニングに使用されたウェブサイトのコンテンツが、潜在顧客がすでに顧客になった時点でサンプリングされており、ウェブサイトには「Websphere」という単語（購入に関するプレス

リリースや顧客が使用する特定の製品機能など)など、購入した IBM 製品の痕跡があったため、漏洩が判明したのです。

Data leakage is subtle and can take many forms. In particular, missing values often represent data leakage.

データの漏えいは微妙であり、様々な形をとることができる。特に、欠損値はデータ漏洩を表すことが多い。

For instance, Jeremy competed in a Kaggle competition designed to predict which researchers would end up receiving research grants. The information was provided by a university and included thousands of examples of research projects, along with information about the researchers involved and data on whether or not each grant was eventually accepted. The university hoped to be able to use the models developed in this competition to rank which grant applications were most likely to succeed, so it could prioritize its processing.

例えば、ジェレミーは、どの研究者が研究助成金を受け取ることになるかを予測するよう設計された Kaggle コンペティションに参加しました。大学から提供された情報には、何千もの研究プロジェクトの例と、関係する研究者情報、各研究助成金が最終的に採択されたかどうかのデータが含まれていました。大学側は、このコンペで開発したモデルを使って、どの助成金申請が成功する可能性が高いかをランク付けし、処理の優先順位をつけられるようにしたいと考えていました。

Jeremy used a random forest to model the data, and then used feature importance to find out which features were most predictive. He noticed three surprising things:

ジェレミーは、ランダムフォレストを使ってデータをモデル化し、特徴の重要度を用いて、どの特徴が最も予測性が高いかを調べました。その結果、3つの驚くべきことがわかりました：

- The model was able to correctly predict who would receive grants over 95% of the time.
このモデルは、95%以上の確率で誰が助成金を受け取るかを正しく予測することができたのです。
- Apparently meaningless identifier columns were the most important predictors.
一見すると無意味な識別子の列が、最も重要な予測因子となった。
- The day of week and day of year columns were also highly predictive; for instance, the vast majority of grant applications dated on a Sunday were accepted, and many accepted grant applications were dated on January 1.

例えば、日曜日に提出された助成金申請書の大半が採択され、採択された助成金申請書の多くが 1 月 1 日付であった。

For the identifier columns, one partial dependence plot per column showed that when the information was missing the application was almost always rejected. It turned out that in practice, the university only filled out much of this information *after* a grant application was accepted. Often, for applications that were not accepted, it was just left blank. Therefore, this information was not something that was actually available at the time that the application was received, and it would not be available for a predictive model—it was data leakage.

識別子の列については、1列につき1つの偏依存プロットがあり、情報が欠落している場合、申請はほとんど却下されることが示された。実際には、大学側がこれらの情報の多くを記入するのは、助成金申請書が受理された後であることが判明した。採択されなかった申請書では、空欄のままになっていることがよくありました。つまり、この情報は、申請書を受理した時点では得られないものであり、予測モデルには利用できない、データリークである。

In the same way, the final processing of successful applications was often done automatically as a batch at the end of the week, or the end of the year. It was this final processing date which ended up in the data, so again, this information, while predictive, was not actually available at the time that the application was received.

同様に、合格したアプリケーションの最終処理は、週明けや年度末に一括して自動処理されることが多かった。この最終処理日がデータとして残ってしまうので、やはり予測可能な情報でありながら、実際には申請書を受け取った時点では入手できなかったのです。

This example showcases the most practical and simple approaches to identifying data leakage, which are to build a model and then:

この例は、データ漏洩を特定するための最も実用的でシンプルなアプローチである、モデルを構築すること、そして次に示すことを紹介しています：

- Check whether the accuracy of the model is *too good to be true*.
モデルの精度が良すぎるかどうかをチェックする。
- Look for important predictors that don't make sense in practice.
実際には意味をなさない重要な予測因子を探す。
- Look for partial dependence plot results that don't make sense in practice.
実際には意味をなさない部分依存のプロット結果を探す。

Thinking back to our bear detector, this mirrors the advice that we provided in <>—it is often a good idea to build a model first and then do your data cleaning, rather than vice versa. The model can help you identify potentially problematic data issues.

クマ発見器のことを考えると、これは<>で提供したアドバイスと同じで、最初にモデルを構築してからデータクリーニングを行うのは、その逆ではなく、しばしば良いアイデアです。モデルは、潜在的に問題のあるデータの問題を特定するのに役立ちます。

It can also help you identify which factors influence specific predictions, with tree interpreters.

また、ツリーインタプリタを使って、どの要因が特定の予測に影響を与えるかを特定するにも役立ちます。

Tree Interpreter

ツリーインタープリター

In []:

```
#hide  
import warnings  
warnings.simplefilter('ignore', FutureWarning)  
from treeinterpreter import treeinterpreter  
from waterfall_chart import plot as waterfall
```

At the start of this section, we said that we wanted to be able to answer five questions:

このセクションの冒頭で、私たちは 5 つの質問に答えられるようになりたいと述べました：

- How confident are we in our predictions using a particular row of data?
あるデータ列を使った予測にどの程度の自信があるのか？
- For predicting with a particular row of data, what were the most important factors, and how did they influence that prediction?
特定の行のデータを使って予測する場合、最も重要な要因は何か、また、その予測にどのような影響を与えたか？
- Which columns are the strongest predictors?
どの列が最も強い予測因子なのか？
- Which columns are effectively redundant with each other, for purposes of prediction?
予測のために、どの列が互いに効果的に冗長になっているか？
- How do predictions vary, as we vary these columns?
これらの列を変化させると、予測はどのように変化するのか？

We've handled four of these already; only the second question remains. To answer this question, we need to use the `treeinterpreter` library. We'll also use the `waterfallcharts` library to draw the chart of the results.

これらのうち 4 つはすでに処理済みで、残るは 2 つの質問だけです。この問い合わせるには、`treeinterpreter` ライブラリを使う必要があります。また、結果のグラフを描くために `waterfallcharts` ライブラリを使うことにします。

```
!pip install treeinterpreter
```

```
!pip install waterfallcharts
```

We have already seen how to compute feature importances across the entire random forest. The basic idea was to look at the contribution of each variable to improving the model, at each branch of every tree, and then add up all of these contributions per variable.

ランダムフォレスト全体の特徴量（feature importances）を計算する方法は既に説明した通りです。その基本的な考え方とは、各ツリーの各ブランチにおいて、モデルを改善するための各変数の寄与度を調べ、変数ごとの寄与度をすべて合計することでした。

We can do exactly the same thing, but for just a single row of data. For instance, let's say we are looking at some particular item at auction. Our model might predict that this item will be very expensive, and we want to know why. So, we take that one row of data and put it through the first decision tree, looking to see what split is used at each point throughout the tree. For each split, we see what the increase or decrease in the addition is, compared to the parent node of the tree. We do this for every tree, and add up the total change in importance by split variable.

これとまったく同じことを、1行のデータに対して行うことができます。例えば、オークションで特定の品物を見ているとします。私たちのモデルは、このアイテムが非常に高価であることを予測し、その理由を知りたいと思います。そこで、この1行のデータを最初の決定木に通し、木全体の各ポイントでどのような分割が使われているかを調べます。それぞれの分割について、ツリーの親ノードとの比較で、加算の増減を確認します。これをすべてのツリーで行い、分割変数による重要度の変化の合計を算出します。

For instance, let's pick the first few rows of our validation set:

例えば、検証セットの最初の数行を選んでみましょう：

In []:

```
row = valid_xs_final.iloc[:5]
```

We can then pass these to treeinterpreter:

これを treeinterpreter に渡します：

In []:

```
prediction,bias,contributions = treeinterpreter.predict(m, row.values)
```

prediction is simply the prediction that the random forest makes. bias is the prediction based on taking the mean of the dependent variable (i.e., the *model* that is the root of every tree). contributions is the most interesting bit—it tells us the total change in prediciton due to each of the independent variables. Therefore, the sum of contributions plus bias must equal the prediction, for each row. Let's look just at the first row:

bias は、従属変数の平均を取ることに基づく予測です（つまり、すべてのツリーのルートとなるモデル）。貢献度は最も興味深いビットで、各独立変数による予測値の総変化を示します。したがって、各行について、寄与とバイアスの合計が予測値に等しくなければなりません。1行目だけを見てみましょう：

In []:

```
prediction[0], bias[0], contributions[0].sum()
```

Out[]:

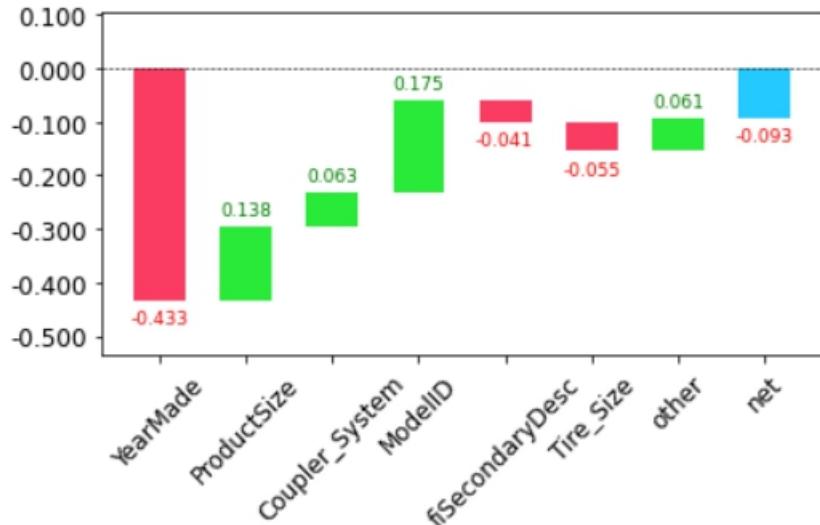
```
(array([10.01216396]), 10.104746057831765, -0.0925820990266335)
```

The clearest way to display the contributions is with a *waterfall plot*. This shows how the positive and negative contributions from all the independent variables sum up to create the final prediction, which is the righthand column labeled "net" here:

寄与を表示する最も明確な方法は、ウォーターフォール図です。これは、すべての独立変数からのプラスとマイナスの寄与が、最終的な予測値を作成するためにどのように合計されるかを示しますが、これはここで「net」とラベルされた右側の列です：

In []:

```
waterfall(valid_xs_final.columns, contributions[0], threshold=0.08,  
         rotation_value=45,formatting='{:,.3f}');
```



This kind of information is most useful in production, rather than during model development. You can use it to provide useful information to users of your data product about the underlying reasoning behind the predictions.

この種の情報は、モデル開発時よりも、むしろ実運用時に最も有用です。これを利用して、データ製品のユーザーに、予測の背後にある根本的な理由について有益な情報を提供することができます。

Now that we covered some classic machine learning techniques to solve this problem, let's see how deep learning can help!

さて、この問題を解決するための古典的な機械学習技術をいくつか取り上げましたが、深層学習がどのように役立つかを見てみましょう！

Extrapolation and Neural Networks

外挿とニューラルネットワーク

A problem with random forests, like all machine learning or deep learning algorithms, is that they don't always generalize well to new data. We will see in which situations neural networks generalize better, but first, let's look at the extrapolation problem that random forests have.

ランダムフォレストの問題点は、他の機械学習や深層学習のアルゴリズムと同様に、新しいデータに対して必ずしもうまく汎化できないことです。どのような場面でニューラルネットワークの方がうまく汎化できるのかを見ていきますが、まずはランダムフォレストが抱える外挿の問題を見ていきましょう。

The Extrapolation Problem

外挿の問題

In []:

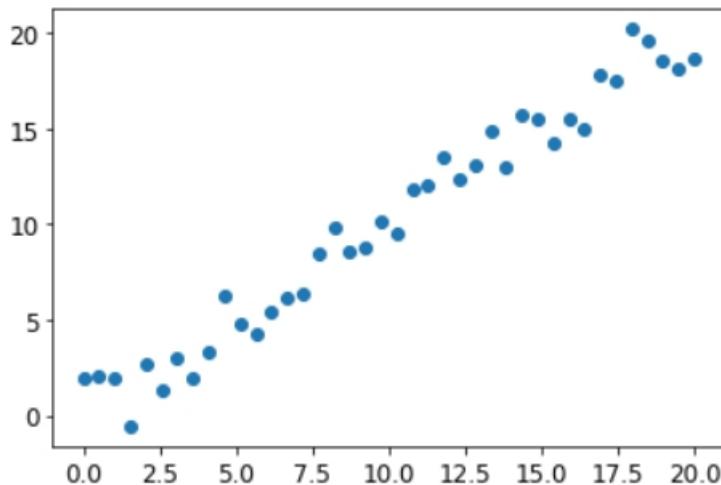
```
#hide  
np.random.seed(42)
```

Let's consider the simple task of making predictions from 40 data points showing a slightly noisy linear relationship:

少しノイズの多い線形関係を示す40点のデータから予測を行うという単純なタスクを考えてみましょう：

In []:

```
x_lin = torch.linspace(0,20, steps=40)  
y_lin = x_lin + torch.randn_like(x_lin)  
plt.scatter(x_lin, y_lin);
```



Although we only have a single independent variable, sklearn expects a matrix of independent variables, not a single vector. So we have to turn our vector into a matrix with one column. In other words, we have to change the *shape* from [40] to [40,1]. One way to do that is with the *unsqueeze* method, which adds a new unit axis to a tensor at the requested dimension:

独立変数は1つだけですが、sklearnは1つのベクトルではなく、独立変数の行列を期待しています。そこで、ベクトルを1列の行列にする必要があります。つまり、[40]から[40,1]に形を変えなければならないのです。これは、テンソルに要求された次元の新しい単位軸

を追加するもので、`unsqueeze` メソッドを使用します：

In []:

```
xs_lin = x_lin.unsqueeze(1)  
x_lin.shape, xs_lin.shape
```

Out[]:

```
(torch.Size([40]), torch.Size([40, 1]))
```

A more flexible approach is to slice an array or tensor with the special value `None`, which introduces an additional unit axis at that location:

より柔軟なアプローチとして、配列やテンソルを特殊な値 `None` でスライスすることで、その位置に追加の単位軸を導入することができます：

In []:

```
x_lin[:, None].shape
```

Out[]:

```
torch.Size([40, 1])
```

We can now create a random forest for this data. We'll use only the first 30 rows to train the model:

このデータに対して、ランダムフォレストを作成することができます。ここでは、最初の30行のみを使用してモデルを訓練することにします：

In []:

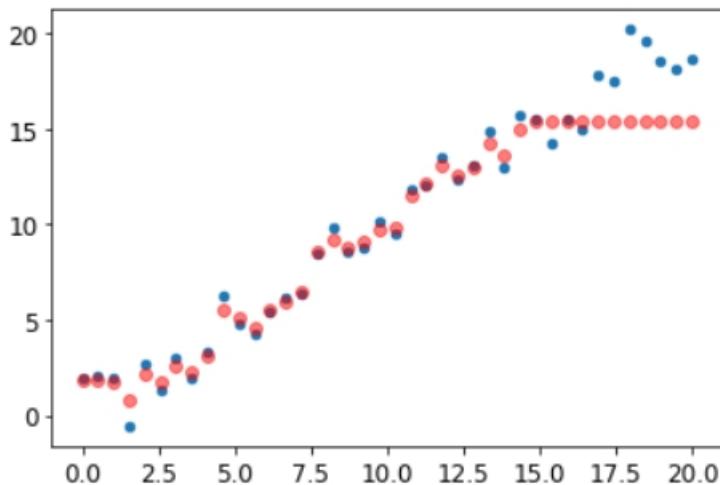
```
m_lin = RandomForestRegressor().fit(xs_lin[:30], y_lin[:30])
```

Then we'll test the model on the full dataset. The blue dots are the training data, and the red dots are the predictions:

次に、フルデータセットでモデルをテストします。青い点がトレーニングデータで、赤い点が予測値です：

In []:

```
plt.scatter(x_lin, y_lin, 20)  
plt.scatter(x_lin, m_lin.predict(xs_lin), color='red', alpha=0.5);
```



We have a big problem! Our predictions outside of the domain that our training data covered are all too low. Why do you suppose this is?

大きな問題があります！学習データがカバーする領域以外での予測値が低すぎます。これはなぜでしょう？

Remember, a random forest just averages the predictions of a number of trees. And a tree simply predicts the average value of the rows in a leaf. Therefore, a tree and a random forest can never predict values outside of the range of the training data. This is particularly problematic for data where there is a trend over time, such as inflation, and you wish to make predictions for a future time. Your predictions will be systematically too low.

ランダムフォレストは、複数の木の予測を平均化したものであることを思い出してください。そして、木は葉の中の行の平均値を予測するだけです。したがって、木やランダムフォレストは、学習データの範囲外の値を予測することはできません。これは、インフレのような経時的なトレンドがあるデータで、将来の時間に対する予測を行いたい場合に特に問題となる。予測値が系統的に低くなりすぎてしまうのです。

But the problem extends beyond time variables. Random forests are not able to extrapolate outside of the types of data they have seen, in a more general sense. That's why we need to make sure our validation set does not contain out-of-domain data.

しかし、この問題は時間変数にとどまりません。ランダムフォレストは、より一般的な意味で、自分が見たことのあるタイプのデータの外側を外挿することができないのです。そのため、検証セットには領域外のデータが含まれていないことを確認する必要があるのです。

Finding Out-of-Domain Data

ドメイン外データの検索

Sometimes it is hard to know whether your test set is distributed in the same way as your training data, or, if it is different, what columns reflect that difference. There's actually an easy way to figure this out, which is to use a

random forest!

テストセットがトレーニングデータと同じように分布しているのか、もし違うのであれば、どの列がその違いを反映しているのかを知ることが難しい場合があります。実はこれを知るには、ランダムフォレストを使うのが簡単な方法です！

But in this case we don't use the random forest to predict our actual dependent variable. Instead, we try to predict whether a row is in the validation set or the training set. To see this in action, let's combine our training and validation sets together, create a dependent variable that represents which dataset each row comes from, build a random forest using that data, and get its feature importance:

しかし、この場合、ランダムフォレストを使って実際の従属変数を予測するわけではありません。その代わりに、ある行が検証セットとトレーニングセットのどちらに含まれるかを予測しようとします。トレーニングセットと検証セットを組み合わせ、それぞれの行がどのデータセットにあるのかを表す従属変数を作成し、そのデータを使ってランダムフォレストを構築し、特徴の重要度を求めてみます：

```
In [ ]:  
df_dom = pd.concat([xs_final, valid_xs_final])  
is_valid = np.array([0]*len(xs_final) + [1]*len(valid_xs_final))  
m = rf(df_dom, is_valid)  
rf_feat_importance(m, df_dom)[:6]
```

Out[]:

	cols	imp
6	saleElapsed	0.891571
9	SalesID	0.091174
14	MachineID	0.012950
0	YearMade	0.001520
10	Enclosure	0.000430
5	ModelID	0.000395

This shows that there are three columns that differ significantly between the training and validation sets: saleElapsed, SalesID, and MachineID. It's fairly obvious why this is the case for saleElapsed: it's the number of days between the start of the dataset and each row, so it directly encodes the date. The difference in SalesID suggests that identifiers for auction sales might increment over time. MachineID suggests something similar might be happening for individual items sold in those auctions.

トレーニングセットと検証セットで大きく異なるのは、SaleElapsed、SalesID、MachineID の 3 列であることがわかる。SaleElapsed は、データセットの開始時点から各行までの日数であり、日付を直接表現しているため、その理由は明らかです。SalesID の違いは、オークション販売の識別子が時間の経過とともに増加する可能性を示唆している。MachineID は、オークションで販売された個々のアイテムについて、同様のことが起こっている可能性を示唆しています。

Let's get a baseline of the original random forest model's RMSE, then see what the effect is of removing each of these columns in turn:

元のランダムフォレストモデルの RMSE の基準値を取得し、これらの列を順番に削除した場合の影響を見てみましょう：

```
In [ ]:  
m = rf(xs_final, y)  
print('orig', m_rmse(m, valid_xs_final, valid_y))  
for c in ('SalesID','saleElapsed','MachineID'):  
    m = rf(xs_final.drop(c,axis=1), y)  
    print(c, m_rmse(m, valid_xs_final.drop(c,axis=1), valid_y))  
  
orig 0.232883  
SalesID 0.230347  
saleElapsed 0.235529  
MachineID 0.230735
```

It looks like we should be able to remove SalesID and MachineID without losing any accuracy. Let's check:

精度を落とさずに SalesID と MachineID を削除することができそうです。確認してみましょう：

```
In [ ]:  
time_vars = ['SalesID','MachineID']  
xs_final_time = xs_final.drop(time_vars, axis=1)  
valid_xs_time = valid_xs_final.drop(time_vars, axis=1)  
m = rf(xs_final_time, y)  
m_rmse(m, valid_xs_time, valid_y)  
  
Out[ ]:  
0.229498
```

Removing these variables has slightly improved the model's accuracy; but more importantly, it should make it more resilient over time, and easier to maintain and understand. We recommend that for all datasets you try building a model where your dependent variable is `is_valid`, like we did here. It can often uncover subtle *domain shift* issues that you may otherwise miss.

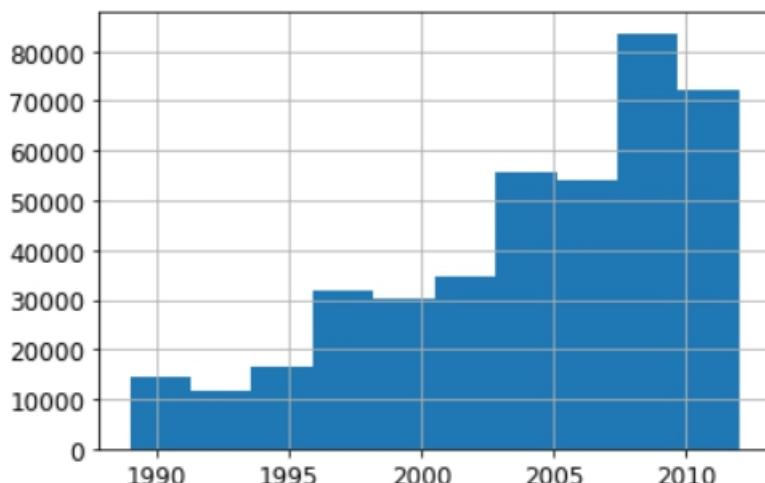
これらの変数を削除することで、モデルの精度は若干向上しましたが、より重要なことは、時間が経っても回復力があり、維持や理解がしやすくなることです。すべてのデータセットで、今回のように従属変数が `is_valid` であるモデルを構築してみることをお勧めします。そうしないと見逃してしまうような、微妙なドメインシフトの問題を発見できることがよくあります。

One thing that might help in our case is to simply avoid using old data. Often, old data shows relationships that just aren't valid any more. Let's try just using the most recent few years of the data:

今回のケースで役に立つかかもしれないのは、単に古いデータを使わないことです。古いデータには、もう有効でない関係が示されていることがあります。そこで、直近の数年分のデータを使ってみることにしましょう：

In []:

```
xs['saleYear'].hist();
```



Here's the result of training on this subset:

このサブセットで学習した結果がこちらです：

In []:

```
filt = xs['saleYear']>2004
xs_filt = xs_final_time[filt]
y_filt = y[filt]
m = rf(xs_filt, y_filt)
m_rmse(m, xs_filt, y_filt), m_rmse(m, valid_xs_time, valid_y)
```

In []:

```
(0.177284, 0.228008)
```

Out[]:

It's a tiny bit better, which shows that you shouldn't always just use your entire dataset; sometimes a subset can be better.

これは、データセット全体を常に使用するのではなく、サブセットを使用した方が良い場合があることを示しています。

Let's see if using a neural network helps.

では、ニューラルネットワークを使うことで効果があるかどうか見てみましょう。

Using a Neural Network

ニューラルネットワークを使う

We can use the same approach to build a neural network model. Let's first replicate the steps we took to set up the TabularPandas object:

同じアプローチでニューラルネットワークモデルを構築することができます。まず、TabularPandas オブジェクトをセットアップするために行った手順を再現してみましょう：

In []:

```
df_nn = pd.read_csv(path/'TrainAndValid.csv', low_memory=False)
df_nn['ProductSize'] = df_nn['ProductSize'].astype('category')
df_nn['ProductSize'].cat.set_categories(sizes, ordered=True, inplace=True)
df_nn[dep_var] = np.log(df_nn[dep_var])
df_nn = add_datepart(df_nn, 'saledate')
```

We can leverage the work we did to trim unwanted columns in the random forest by using the same set of columns for our neural network:

ランダムフォレストで不要な列を削るために行った作業を、ニューラルネットワークに同じ列のセットを使用することで活用できます：

In []:

```
df_nn_final = df_nn[list(xs_final_time.columns) + [dep_var]]
```

Categorical columns are handled very differently in neural networks, compared to decision tree approaches. As we saw in <<chapter_collab>>, in a neural net a great way to handle categorical variables is by using embeddings. To create embeddings, fastai needs to determine which columns should be treated as categorical variables. It does this by comparing the number of distinct levels in the variable to the value of the max_card parameter. If it's lower, fastai will treat the variable as categorical. Embedding sizes larger than 10,000 should generally only be used after you've tested whether there are better ways to group the variable, so we'll use 9,000 as our max_card:

ニューラルネットでは、決定木のアプローチと比較して、カテゴリーカラムの扱いが大きく異なります。chapter_collab>>で見たように、ニューラルネットではカテゴリ変数を扱

うのにエンベッディングを使うのが良い方法です。エンベッディングを作成するために、fastai はどのカラムをカテゴリー変数として扱うべきかを決定する必要があります。これは、変数のレベル数を `max_card` パラメータの値と比較することで行います。もしそれ以下であれば、fastai はその変数をカテゴリカルとして扱います。10,000 を超える埋め込みサイズは、一般に、変数をグループ化するためのより良い方法があるかどうかをテストした後にのみ使用されるべきですので、ここでは 9,000 を `max_card` として使用します：

In []:

```
cont_nn,cat_nn = cont_cat_split(df_nn_final, max_card=9000, dep_var=dep_var)
```

In this case, there's one variable that we absolutely do not want to treat as categorical: the `saleElapsed` variable. A categorical variable cannot, by definition, extrapolate outside the range of values that it has seen, but we want to be able to predict auction sale prices in the future. Let's verify that `cont_cat_split` did the correct thing.

この場合、絶対にカテゴリカルとして扱いたくない変数が 1 つあります：`saleElapsed` 変数です。カテゴリカル変数は、定義上、その変数が見た値の範囲外を外挿することはできませんが、私たちは将来のオークション販売価格を予測できるようにしたいのです。

`cont_cat_split` が正しいことをしたかどうか、検証してみよう。

In []:

```
cont_nn
```

Out[]:

```
['saleElapsed']
```

Let's take a look at the cardinality of each of the categorical variables that we have chosen so far:

ここまでで選んだ各カテゴリカル変数のカーディナリティを見てみましょう：

In []:

```
df_nn_final[cat_nn].nunique()
```

Out[]:

YearMade	73
ProductSize	6
Coupler_System	2
fiProductClassDesc	74
Hydraulics_Flow	3
ModelID	5281
fiSecondaryDesc	177
fiModelDesc	5059
Enclosure	6
Hydraulics	12
ProductGroup	6

```
Drive_System          4  
Tire_Size            17  
dtype: int64
```

The fact that there are two variables pertaining to the "model" of the equipment, both with similar very high cardinalities, suggests that they may contain similar, redundant information. Note that we would not necessarily see this when analyzing redundant features, since that relies on similar variables being sorted in the same order (that is, they need to have similarly named levels). Having a column with 5,000 levels means needing 5,000 columns in our embedding matrix, which would be nice to avoid if possible. Let's see what the impact of removing one of these model columns has on the random forest:

機器の「モデル」に関する変数が 2 つあり、どちらも同じような非常に高いカーディナリティを持つという事実は、これらの変数に類似の冗長な情報が含まれている可能性を示唆している。冗長な特徴を分析する場合、同じような変数が同じ順序でソートされる（つまり、同じような名前のレベルを持つ必要がある）ことに依存するため、必ずしもこれを見ることはできないことに注意してください。5,000 のレベルを持つ列を持つことは、埋め込み行列に 5,000 の列を必要とすることを意味し、可能であれば避けたいところである。これらのモデルカラムを 1 つ削除することが、ランダムフォレストにどのような影響を与えるか見てみましょう：

```
In [ ]:  
xs_filt2 = xs_filt.drop('fiModelDescriptor', axis=1)  
valid_xs_time2 = valid_xs_time.drop('fiModelDescriptor', axis=1)  
m2 = rf(xs_filt2, y_filt)  
m_rmse(m2, xs_filt2, y_filt), m_rmse(m2, valid_xs_time2, valid_y)  
Out[ ]:  
(0.176713, 0.230195)
```

There's minimal impact, so we will remove it as a predictor for our neural network:

影響は少ないので、ニューラルネットワークの予測因子として削除します：

```
In [ ]:  
cat_nn.remove('fiModelDescriptor')
```

We can create our TabularPandas object in the same way as when we created our random forest, with one very important addition: normalization. A random forest does not need any normalization—the tree building procedure cares only about the order of values in a variable, not at all about how they are scaled. But as we have seen, a neural network definitely does care about this. Therefore, we add the Normalize processor when we build our TabularPandas object:

TabularPandas オブジェクトは、ランダムフォレストを作成したときと同じ方法で作成できますが、1 つだけ非常に重要な追加事項があります：正規化です。ランダムフォレストで

は正規化は必要ありません。ツリー構築の手順では、変数の値の順序だけを気にし、どのようにスケーリングされるかはまったく気にしません。しかし、これまで見てきたように、ニューラルネットワークは間違いなくこの点を気にします。そこで、TabularPandas オブジェクトを構築する際に、Normalize プロセッサを追加しています：

In []:

```
procs_nn = [Categorify, FillMissing, Normalize]
to_nn = TabularPandas(df_nn_final, procs_nn, cat_nn, cont_nn,
                      splits=splits, y_names=dep_var)
```

Tabular models and data don't generally require much GPU RAM, so we can use larger batch sizes:

表形式のモデルやデータは一般的に GPU RAM をあまり必要としないので、より大きなバッチサイズを使用することができます：

In []:

```
dls = to_nn.dataloaders(1024)
```

As we've discussed, it's a good idea to set `y_range` for regression models, so let's find the min and max of our dependent variable:

これまで述べてきたように、回帰モデルには `y_range` を設定するのがよいので、従属変数の最小値と最大値を求めてみましょう：

In []:

```
y = to_nn.train.y
y.min(), y.max()
```

Out[]:

```
(8.465899467468262, 11.863582611083984)
```

We can now create the Learner to create this tabular model. As usual, we use the application-specific learner function, to take advantage of its application-customized defaults. We set the loss function to MSE, since that's what this competition uses.

これで、この表形式モデルを作成するための Learner を作成できます。いつものように、アプリケーション固有の学習器関数を使用し、アプリケーションでカスタマイズされたデフォルトを活用します。損失関数は MSE に設定します。この競技では MSE を使うからです。

By default, for tabular data fastai creates a neural network with two hidden layers, with 200 and 100 activations, respectively. This works quite well for small datasets, but here we've got quite a large dataset, so we increase the layer sizes to 500 and 250:

デフォルトでは、表形式のデータに対して、fastai は 2 つの隠れ層を持つニューラルネットワークを作成し、それぞれ 200 と 100 の活性度を持つ。これは小さなデータセットでは非常に有効ですが、今回はかなり大きなデータセットなので、レイヤーサイズを 500 と 250

に増やします:

In []:

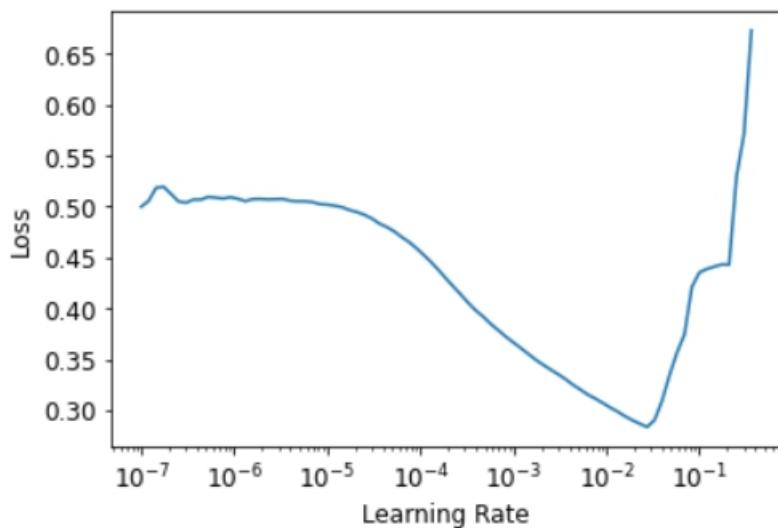
```
learn = tabular_learner(dls, y_range=(8,12), layers=[500,250],  
n_out=1, loss_func=F.mse_loss)
```

In []:

```
learn.lr_find()
```

Out[]:

```
SuggestedLRs(lr_min=0.002754228748381138, lr_stEEP=0.00015848931798  
245758)
```



There's no need to use `fine_tune`, so we'll train with `fit_one_cycle` for a few epochs and see how it looks:

`fine_tune` を使う必要はないので、`fit_one_cycle` で数エポック訓練して様子を見ることにします:

In []:

```
learn.fit_one_cycle(5, 1e-2)
```

epoch	train_loss	valid_loss	time
0	0.068459	0.061185	00:09
1	0.056469	0.058471	00:09
2	0.048689	0.052404	00:09
3	0.044529	0.052138	00:09
4	0.040860	0.051236	00:09

We can use our `r_mse` function to compare the result to the random forest result we got earlier:

`r_mse` 関数を使って、先ほどのランダムフォレストの結果と比較することができます：

In []:

```
preds,targs = learn.get_preds()  
r_mse(preds,targs)
```

Out[]:

0.226353

It's quite a bit better than the random forest (although it took longer to train, and it's fussier about hyperparameter tuning).

ランダムフォレストよりもかなり優れています（ただし、訓練に時間がかかり、ハイパーパラメータのチューニングに手間がかかります）。

Before we move on, let's save our model in case we want to come back to it again later:

次に進む前に、後でまた戻ってきたくなるように、モデルを保存しておきましょう：

In []:

```
learn.save('nn')
```

Out[]:

Path('models/nn.pth')

Sidebar: fastai's Tabular Classes

サイドバー: fastai の Tabular クラス

In fastai, a tabular model is simply a model that takes columns of continuous or categorical data, and predicts a category (a classification model) or a continuous value (a regression model). Categorical independent variables are passed through an embedding, and concatenated, as we saw in the neural net we used for collaborative filtering, and then continuous variables are concatenated as well.

fastai では、表モデルとは、連続またはカテゴリーデータを列として受け取り、カテゴリーモデル（分類モデル）または連続値（回帰モデル）を予測するモデルのことである。カテゴリ型の独立変数は、協調フィルタリングを使ったニューラルネットで見たように、エンベッディングを通過して連結され、さらに連続変数も連結される。

The model created in `tabular_learner` is an object of class `TabularModel`. Take a look at the source for `tabular_learner` now (remember, that's `tabular_learner??` in Jupyter). You'll see that like `collab_learner`, it first calls `get_emb_sz` to calculate appropriate embedding sizes (you can override these by using the `emb_szs` parameter, which is a dictionary containing any column names you want to set sizes for manually), and it sets a few other defaults. Other than that, it just creates the `TabularModel`, and passes that to `TabularLearner` (note that `TabularLearner` is identical to `Learner`, except for a customized `predict` method).

`tabular_learner` で作成されるモデルは、`TabularModel` というクラスのオブジェクトです。今すぐ `tabular_learner` のソースを見てみてください（Jupyter の `tabular_learner?` を思い出してください）。`collab_learner` と同様に、まず `get_emb_sz` を呼び出して適切な埋め込みサイズを計算し（`emb_szs` パラメータを使うことでこれらをオーバーライドできます、これは手動でサイズを設定したい列名を含む辞書です）、いくつかの他のデフォルトを設定していることがわかるでしょう。それ以外は `TabularModel` を作成し、それを `TabularLearner` に渡すだけです（`TabularLearner` はカスタマイズされた `predict` メソッドを除いて `Learner` と同じであることに注意してください）。

That means that really all the work is happening in `TabularModel`, so take a look at the source for that now. With the exception of the `BatchNorm1d` and `Dropout` layers (which we'll be learning about shortly), you now have the knowledge required to understand this whole class. Take a look at the discussion of `EmbeddingNN` at the end of the last chapter. Recall that it passed `n_cont=0` to `TabularModel`. We now can see why that was: because there are zero continuous variables (in fastai the `n_` prefix means "number of," and `cont` is an abbreviation for "continuous").

つまり、本当にすべての作業は `TabularModel` で行われているのです。そこで、そのソースをご覧ください。`BatchNorm1d` と `Dropout` レイヤーを除いて（このレイヤーについてはまもなく学習します）、このクラス全体を理解するのに必要な知識はこれで揃いました。前章の最後にある `EmbeddingNN` の議論を見てみよう。その際、`TabularModel` に `n_cont=0` を渡していたことを思い出してください。その理由は、連続変数がゼロだからです（fastai では `n_` という接頭辞は「数」の意味で、`cont` は「連続」の省略形です）。

End sidebar

サイドバーを閉じる

Another thing that can help with generalization is to use several models and average their predictions—a technique, as mentioned earlier, known as *ensembling*.

また、汎化のためには、複数のモデルを用いて、その予測値を平均化することが有効である。

Ensembling

アンサンブル

Think back to the original reasoning behind why random forests work so well: each tree has errors, but those errors are not correlated with each other, so the average of those errors should tend towards zero once there are enough trees. Similar reasoning could be used to consider averaging the predictions of models trained using different algorithms.

ランダムフォレストがなぜうまく機能するのか、その元となった理由を思い返してみてください。同様の理由で、異なるアルゴリズムで学習したモデルの予測値を平均化することができます。

In our case, we have two very different models, trained using very different algorithms: a random forest, and a neural network. It would be reasonable to expect that the kinds of errors that each one makes would be quite different. Therefore, we might expect that the average of their predictions would be better than either one's individual predictions.

今回のケースでは、ランダムフォレストとニューラルネットワークという、非常に異なるアルゴリズムで学習させた2つのモデルがあります。それぞれで発生する誤差の種類はかなり異なると考えるのが妥当でしょう。したがって、それぞれの予測値の平均は、どちらか一方の予測値よりも優れていると予想されます。

As we saw earlier, a random forest is itself an ensemble. But we can then include a random forest in *another* ensemble—an ensemble of the random forest and the neural network! While ensembling won't make the difference between a successful and an unsuccessful modeling process, it can certainly add a nice little boost to any models that you have built.

先に見たように、ランダムフォレストはそれ自体がアンサンブルである。しかし、ランダムフォレストを別のアンサンブル、つまりランダムフォレストとニューラルネットワークのアンサンブルに含めることができます！アンサンブルは、モデリングプロセスの成否を分けるものではありませんが、構築したモデルに良い刺激を与えることは確かです。

One minor issue we have to be aware of is that our PyTorch model and our sklearn model create data of different types: PyTorch gives us a rank-2 tensor (i.e., a column matrix), whereas NumPy gives us a rank-1 array (a vector). `squeeze` removes any unit axes from a tensor, and `to_np` converts it into a NumPy array:

PyTorchのモデルとsklearnのモデルでは、作成するデータの種類が異なることに注意する必要があります：PyTorchはランク2のテンソル（つまり列の行列）、NumPyはランク1の配列（ベクトル）です。`squeeze`はテンソルから単位軸を取り除き、`to_np`はそれをNumPyの配列に変換します：

In []:

```
rf_preds = m.predict(valid_xs_time)
ens_preds = (to_np(preds.squeeze()) + rf_preds) / 2
```

This gives us a better result than either model achieved on its own:

この結果、どちらのモデルも単独で達成した結果よりも良い結果が得られました：

In []:

```
r_mse(ens_preds, valid_y)
```

Out[]:

0.222134

In fact, this result is better than any score shown on the Kaggle leaderboard. It's not directly comparable, however, because the Kaggle leaderboard uses a separate dataset that we do not have access to. Kaggle does not allow us to submit to this old competition to find out how we would have done, but our results certainly look very encouraging!

実際、この結果は Kaggle のリーダーボードに表示されるどのスコアよりも優れています。しかし、Kaggle のリーダーボードは、私たちがアクセスできない別のデータセットを使用しているため、直接比較することはできません。Kaggle では、この古いコンペティションに参加して、私たちならどうだったかを調べることはできませんが、私たちの結果が非常に有望であることは確かです！

Boosting

ブースティング

So far our approach to ensembling has been to use *bagging*, which involves combining many models (each trained on a different data subset) together by averaging them. As we saw, when this is applied to decision trees, this is called a *random forest*.

これまで、アンサンブルのアプローチとして、バギング（Bagging）を使ってきました。バギングとは、多くのモデル（それぞれが異なるデータサブセットで学習したもの）を平均化することで結合することです。バギングを決定木に適用すると、ランダムフォレストと呼ばれるようになることはお分かりいただけたと思う。

There is another important approach to ensembling, called *boosting*, where we add models instead of averaging them. Here is how boosting works:

アンサンブルにはもう一つ重要なアプローチがあり、ブースティングと呼ばれるもので、平均化する代わりにモデルを追加するものです。ブースティングの仕組みは以下の通りです：

- Train a small model that underfits your dataset.
データセットに適合しない小さなモデルをトレーニングする。
- Calculate the predictions in the training set for this model.
このモデルのトレーニングセットにおける予測値を計算する。
- Subtract the predictions from the targets; these are called the "residuals" and represent the error for each point in the training set.
ターゲットから予測値を引きます。これは「残差」と呼ばれ、トレーニングセットの各ポイントの誤差を表しています。
- Go back to step 1, but instead of using the original targets, use the residuals as the targets for the training.
ステップ 1 に戻るが、元のターゲットを使う代わりに、残差をトレーニングのターゲットとして使う。

- Continue doing this until you reach some stopping criterion, such as a maximum number of trees, or you observe your validation set error getting worse.
これを、木の最大本数などの停止基準に達するか、検証セットの誤差が悪化するのを確認するまで続けます。

Using this approach, each new tree will be attempting to fit the error of all of the previous trees combined. Because we are continually creating new residuals, by subtracting the predictions of each new tree from the residuals from the previous tree, the residuals will get smaller and smaller.

このアプローチでは、新しいツリーは、以前のツリーの誤差をすべて合わせたものに適合させようとします。常に新しい残差を作っているので、前の木の残差から各新しい木の予測値を引くことで、残差はどんどん小さくなっています。

To make predictions with an ensemble of boosted trees, we calculate the predictions from each tree, and then add them all together. There are many models following this basic approach, and many names for the same models. *Gradient boosting machines* (GBMs) and *gradient boosted decision trees* (GBDTs) are the terms you're most likely to come across, or you may see the names of specific libraries implementing these; at the time of writing, *XGBoost* is the most popular.

ブーストツリーのアンサンブルで予測を行うには、各ツリーの予測値を計算し、それらをすべて足し合わせます。このような基本的なアプローチをとるモデルは数多く存在し、同じモデルでも多くの名前があります。勾配ブースティング機械（GBM）や勾配ブースティング決定木（GBDT）などがよく目にする用語で、これらを実装した特定のライブラリの名前を目にすることもあるはずだ。

Note that, unlike with random forests, with this approach there is nothing to stop us from overfitting. Using more trees in a random forest does not lead to overfitting, because each tree is independent of the others. But in a boosted ensemble, the more trees you have, the better the training error becomes, and eventually you will see overfitting on the validation set.

ランダムフォレストとは異なり、このアプローチではオーバーフィッティングを止めるものが無いことに注意してください。ランダムフォレストでは、より多くの木を使っても、オーバーフィッティングは起こりません。しかし、ブースト・アンサンブルでは、木が多いほど学習誤差が大きくなり、最終的には検証セットでオーバーフィッティングが発生します。

We are not going to go into detail on how to train a gradient boosted tree ensemble here, because the field is moving rapidly, and any guidance we give will almost certainly be outdated by the time you read this. As we write this, sklearn has just added a `HistGradientBoostingRegressor` class that provides excellent performance. There are many hyperparameters to tweak for this class, and for all gradient boosted tree methods we have seen. Unlike random forests, gradient boosted trees are extremely sensitive to the choices of these hyperparameters; in practice, most people use a loop that tries a range of different hyperparameters to find the ones that work best.

なぜなら、この分野は急速に発展しており、この記事を読む頃には、私たちが提供する情報はほとんど時代遅れになっていることでしょう。これを書いている間に、sklearn は `HistGradientBoostingRegressor` クラスを追加し、優れたパフォーマンスを提供するようになりました。このクラスや、これまで見てきたすべての勾配ブーストツリー法には、調整すべきハイパーパラメータがたくさんあります。ランダムフォレストとは異なり、勾配ブースト木はこれらのハイパーパラメータの選択に非常に敏感です。実際には、ほとんどの人が、さまざまなハイパーパラメータを試して、最もうまくいくものを見つけるループを使用します。

One more technique that has gotten great results is to use embeddings learned by a neural net in a machine learning model.

もうひとつ、機械学習モデルでニューラルネットが学習したエンベッディングを使用することで、素晴らしい結果を得ることができます。

Combining Embeddings with Other Methods

エンベッディングと他の手法の組み合わせ

The abstract of the entity embedding paper we mentioned at the start of this chapter states: "the embeddings obtained from the trained neural network boost the performance of all tested machine learning methods considerably when used as the input features instead". It includes the very interesting table in <>.

本章の冒頭で紹介したエンティティエンベッディングの論文の要旨には、次のように書かれています： "訓練されたニューラルネットワークから得られた埋め込みは、代わりに入力特徴として使用された場合、テストされたすべての機械学習方法の性能をかなり高める" とあります。この論文には、<>にある非常に興味深い表が含まれています。

method	MAPE	MAPE (with EE)
KNN	0.290	0.116
random forest	0.158	0.108
gradient boosted trees	0.152	0.115
neural network	0.101	0.093

エンベッディングと他の手法の組み合わせ

This is showing the mean average percent error (MAPE) compared among four different modeling techniques, three of which we have already seen, along with k -nearest neighbors (KNN), which is a very simple baseline method. The first numeric column contains the results of using the methods on the data provided in the competition; the second column shows what happens if you first train a neural network with categorical embeddings, and then use those categorical embeddings instead of the raw categorical columns in the model. As you see, in every case, the models are dramatically improved by using the embeddings instead of the raw categories.

これは、4つの異なるモデリング手法の平均平均パーセントエラー（MAPE）を比較したもので、すでに見た3つの手法と、非常にシンプルなベースライン手法である KNN(k -nearest neighbors) が含まれています。最初の数値列は、コンペで提供されたデータに対して各手法を使用した結果です。2番目の列は、まずカテゴリ埋め込みでニューラルネットワークを訓練し、モデル内の生のカテゴリ列の代わりにそのカテゴリ埋め込みを使用するとどうなるかを示しています。ご覧のように、どのケースでも、生のカテゴリの代わりにエンベッディングを使うことで、モデルは劇的に改善されていることがわかります。

This is a really important result, because it shows that you can get much of the performance improvement of a neural network without actually having to use a neural network at inference time. You could just use an embedding, which is literally just an array lookup, along with a small decision tree ensemble.

これは、推論時にニューラルネットワークを使わなくても、ニューラルネットワークの性能を大幅に向上させることができることを示すもので、非常に重要な結果です。エンベッディングは、文字通り配列のルックアップであり、小さな決定木アンサンブルと一緒に使うだけよいのです。

These embeddings need not even be necessarily learned separately for each model or task in an organization. Instead, once a set of embeddings are learned for some column for some task, they could be stored in a central place, and reused across multiple models. In fact, we know from private communication with other practitioners at large companies that this is already happening in many places.

このエンベッディングは、必ずしも組織内の各モデルやタスクごとに別々に学習する必要もない。その代わり、あるタスクのあるカラムについて一旦エンベッディングを学習したら、それを中央の場所に保存し、複数のモデルで再利用することができます。実際、私たちは、大企業の実務担当者との個人的なコミュニケーションから、このようなことがすでに多くの場所で行われていることを知りました。

Conclusion: Our Advice for Tabular Modeling

おわりに タブラー モデリングへのアドバイス

We have discussed two approaches to tabular modeling: decision tree ensembles and neural networks. We've also mentioned two different decision tree ensembles: random forests, and gradient boosting machines. Each is very effective, but each also has compromises:

これまで、表形式モデリングには、決定木アンサンブルとニューラルネットワークという2つのアプローチがあることを説明してきました。また、ランダムフォレストとグラディエントブースティングマシンという2つの異なる決定木アンサンブルについて説明しました。それぞれ非常に効果的ですが、妥協点もあります：

Random forests are the easiest to train, because they are extremely resilient to hyperparameter choices and require very little preprocessing. They are very fast to train, and should not overfit if you have enough trees. But they can be a little less accurate, especially if extrapolation is required, such as predicting future time periods.

ランダムフォレストは、ハイパーパラメータの選択に非常に強く、前処理をほとんど必要としないため、最も簡単に訓練することができます。ランダムフォレストは、ハイパーパラメータの選択に非常に強く、前処理がほとんど必要ないため、訓練が非常に速く、十分な数の木があればオーバーフィットすることはありません。しかし、特に将来の時間帯を予測するような外挿が必要な場合は、精度が少し落ちることがあります。

Gradient boosting machines in theory are just as fast to train as random forests, but in practice you will have to try lots of different hyperparameters. They can overfit, but they are often a little more accurate than random forests.

勾配ブースティング・マシンは、理論的にはランダムフォレストと同様に高速に学習できますが、実際には多くの異なるハイパーパラメータを試す必要があります。オ

一バーフィットすることもありますが、ランダムフォレストよりも若干精度が高いことが多いようです。

Neural networks take the longest time to train, and require extra preprocessing, such as normalization; this normalization needs to be used at inference time as well. They can provide great results and extrapolate well, but only if you are careful with your hyperparameters and take care to avoid overfitting.

ニューラルネットワークは、学習に最も時間がかかり、正規化などの前処理が必要です。この正規化は、推論時にも使用する必要があります。しかし、ハイパーパラメータに注意し、オーバーフィッティングを避けることができれば、素晴らしい結果が得られ、外挿もうまくいきます。

We suggest starting your analysis with a random forest. This will give you a strong baseline, and you can be confident that it's a reasonable starting point. You can then use that model for feature selection and partial dependence analysis, to get a better understanding of your data.

ランダムフォレストから分析を始めることをお勧めします。そうすることで、強力なベースラインができ、それが妥当な出発点であることを確信できます。その後、そのモデルを使って特徴選択と部分依存性分析を行い、データの理解を深めることができます。

From that foundation, you can try neural nets and GBMs, and if they give you significantly better results on your validation set in a reasonable amount of time, you can use them. If decision tree ensembles are working well for you, try adding the embeddings for the categorical variables to the data, and see if that helps your decision trees learn better.

その基礎の上で、ニューラルネットや GBM を試してみて、妥当な時間で検証セットで有意に良い結果が得られるなら、それを使うことができます。決定木アンサンブルでうまくいっている場合は、カテゴリ変数の埋め込みをデータに追加してみて、決定木の学習がうまくいくかどうか確認してみてください。

Questionnaire

質問票

1. What is a continuous variable?

連続変数とは？

2. What is a categorical variable?

カテゴリカル変数とは何か？

3. Provide two of the words that are used for the possible values of a categorical variable.

カテゴリカル変数の取り得る値に対して使われる言葉を 2 つ挙げてください。

4. What is a "dense layer"?

密な層」とは何ですか？

5. How do entity embeddings reduce memory usage and speed up neural networks?

エンティティエンベッディングはどのようにしてメモリ使用量を減らし、ニューラルネットワークを高速化するのか？

6. What kinds of datasets are entity embeddings especially useful for?

エンティティエンベッディングは、どのようなデータセットに特に有効か？

7. What are the two main families of machine learning algorithms?

機械学習アルゴリズムの2つの主要なファミリーは何ですか？

8. Why do some categorical columns need a special ordering in their classes? How do you do this in Pandas?
なぜ、あるカテゴリカルな列は、そのクラスにおいて特別な順序を必要とするのでしょうか？Pandasでこれを行うにはどうすればいいのか？

9. Summarize what a decision tree algorithm does.

決定木アルゴリズムが何をするのか要約してください。

10. Why is a date different from a regular categorical or continuous variable, and how can you preprocess it to allow it to be used in a model?

日付が通常のカテゴリ変数や連続変数と異なる理由と、それをモデルで使用できるようにするために前処理は？

11. Should you pick a random validation set in the bulldozer competition? If no, what kind of validation set should you pick?

ブルドーザー競技では、ランダムな検証セットを選ぶべきですか？もしそうでなければ、どのような検証セットを選ぶべきですか？

12. What is pickle and what is it useful for?

pickleとは何ですか、また何に役立つのですか？

13. How are mse, samples, and values calculated in the decision tree drawn in this chapter?

この章で描いた決定木では、mse、サンプル、値はどのように計算されるのか？

14. How do we deal with outliers, before building a decision tree?

決定木を作成する前に、外れ値をどのように扱うか？

15. How do we handle categorical variables in a decision tree?

決定木の中でカテゴリ変数をどのように扱うか？

16. What is bagging?

バギングとは何ですか？

17. What is the difference between max_samples and max_features when creating a random forest?

ランダムフォレストを作成する際のmax_samplesとmax_featuresの違いは何ですか？

18. If you increase n_estimators to a very high value, can that lead to overfitting? Why or why not?

n_estimatorsを非常に大きな値にした場合、オーバーフィッティングになる可能性はありますか？なぜでしょうか、それともそうではないのでしょうか？

19. In the section "Creating a Random Forest", just after <>max_features>>, why did preds.mean(0) give the

same result as our random forest?

「ランダムフォレストの作成」セクションの<>のすぐ後にある `preds.mean(0)`が、私たちのランダムフォレストと同じ結果を与えたのはなぜですか？

20. What is "out-of-bag-error"?

「アウトオブバグエラー」とは何ですか？

21. Make a list of reasons why a model's validation set error might be worse than the OOB error. How could you test your hypotheses?

モデルの検証セットの誤差が OOB 誤差より悪いかもしれない理由をリストアップしてください。その仮説はどのように検証するのか？

22. Explain why random forests are well suited to answering each of the following questions:

ランダムフォレストが以下の各質問に答えるのに適している理由を説明しなさい：

- How confident are we in our predictions using a particular row of data?

特定のデータ行を使った予測にどの程度の自信があるか？

- For predicting with a particular row of data, what were the most important factors, and how did they influence that prediction?

ある特定の行のデータを使って予測する場合、最も重要な要因は何か、そしてその予測にどのような影響を与えたか？

- Which columns are the strongest predictors?

どの列が最も強い予測因子なのか？

- How do predictions vary as we vary these columns?

これらの列を変化させると、予測はどのように変化するのか？

23. What's the purpose of removing unimportant variables?

重要でない変数を除去する目的は何ですか？

24. What's a good type of plot for showing tree interpreter results?

ツリーインタープリターの結果を表示するのに適したプロットのタイプは何ですか？

25. What is the "extrapolation problem"?

「外挿問題」とは何ですか？

26. How can you tell if your test or validation set is distributed in a different way than your training set?

テストセットや検証セットがトレーニングセットと異なる方法で分布しているかどうかは、どのように見分けることができますか？

27. Why do we ensure `saleElapsed` is a continuous variable, even although it has less than 9,000 distinct

values?

`SaleElapsed` の値が 9,000 未満であるにもかかわらず、連続変数であることを確認するのはなぜですか？

28. What is "boosting"?

「ブースティング」とは何ですか？

29. How could we use embeddings with a random forest? Would we expect this to help?

ランダムフォレストでエンベディングを使うのはどうだろう？これは役に立つのだろうか？

30. Why might we not always use a neural net for tabular modeling?

表形式のモデリングにニューラルネットを使用するとは限らないのはなぜか？

Further Research

さらなる研究

1. Pick a competition on Kaggle with tabular data (current or past) and try to adapt the techniques seen in this chapter to get the best possible results. Compare your results to the private leaderboard.

Kaggle で表形式データ（現在または過去）のあるコンペティションを選び、この章で見たテクニックを適応して、可能な限り最高の結果を得るようにしてください。自分の結果をプライベートリーダーボードと比較する。

2. Implement the decision tree algorithm in this chapter from scratch yourself, and try it on the dataset you used in the first exercise.

本章の決定木アルゴリズムをゼロから実装し、最初の演習で使用したデータセットで試してみる。

3. Use the embeddings from the neural net in this chapter in a random forest, and see if you can improve on the random forest results we saw.

この章のニューラルネットの埋め込みをランダムフォレストに使用し、私たちが見たランダムフォレストの結果を改善できるかどうかを確認する。

4. Explain what each line of the source of TabularModel does (with the exception of the BatchNorm1d and Dropout layers).

TabularModel のソースの各行が何をしているのか説明しなさい（BatchNorm1d と Dropout レイヤーを除く）。