Course 2 – Machine Learning Data Lifecycle in Production

In the second course of Machine Learning Engineering for Production Specialization, you will build data pipelines by gathering, cleaning, and validating datasets and assessing data quality; implement feature engineering, transformation, and selection with TensorFlow Extended and get the most predictive power out of your data; and establish the data lifecycle by leveraging data lineage and provenance metadata tools and follow data evolution with enterprise data schemas.

Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

Week 2: Feature Engineering, Transformation and Selection

Contents

Introduction to Preprocessing

> *"Coming up with features is difficult, time-consuming, and requires expert knowledge. Applied machine learning often requires careful engineering of the features and dataset."*
> — Andrew Ng

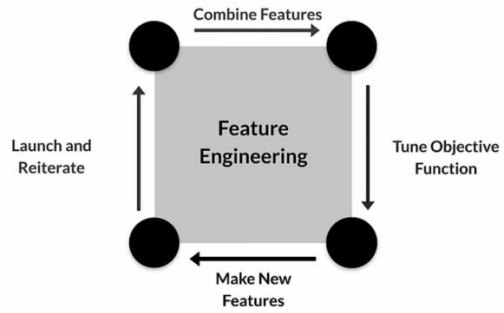
- Welcome to feature engineering, transformation and selection. Now, some of this material may seem like it's review, especially from earlier courses or if you've worked in an academic or research setting.
- But here, we're going to be focusing on production issues, one of which means being able to do all of this at scale in a reproducible and consistent way, so let's get started.

Squeezing the most out of data

- Making data useful before training a model
- Representing data in forms that help models learn
- Increasing predictive quality
- Reducing dimensionality with feature engineering

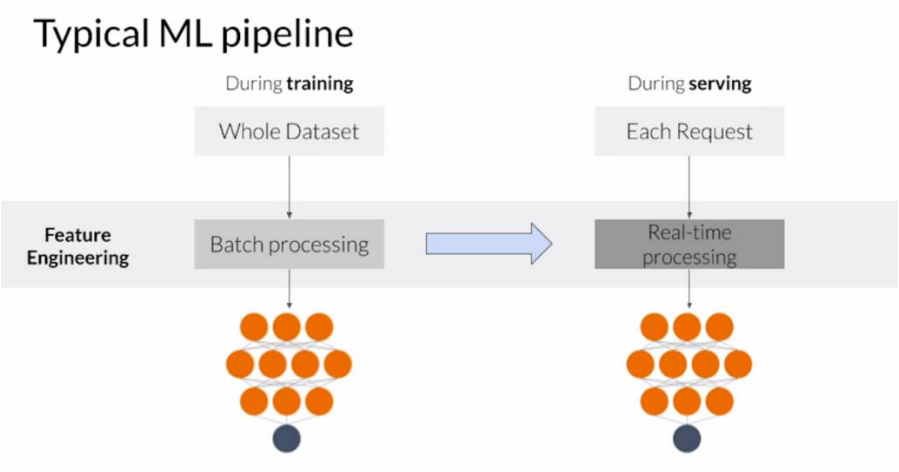
- So let's start with how we're going to squeeze information really out of our data. So, ML models usually require some data pre-processing to improve training and you should probably have been training models by now, and this should be very familiar to you.
- What may not be quite so familiar are some of the issues that are involved in production environments, so that's where we'll focus.
- The way that data is represented can really have a big influence on how a model is able to learn from it. For example, models tend to converge much faster and more reliably when numerical data has been normalized.
- So techniques for selecting and transforming the input data are key to increase predictive quality of models
- Dimensionality reduction is recommended whenever possible, so that the most relevant information is preserved, while both the representation and prediction ability are enhanced, and the required compute resources are reduced.
- Remember, in production ML compute resources are a key contributor to the cost of running a model

Art of feature engineering

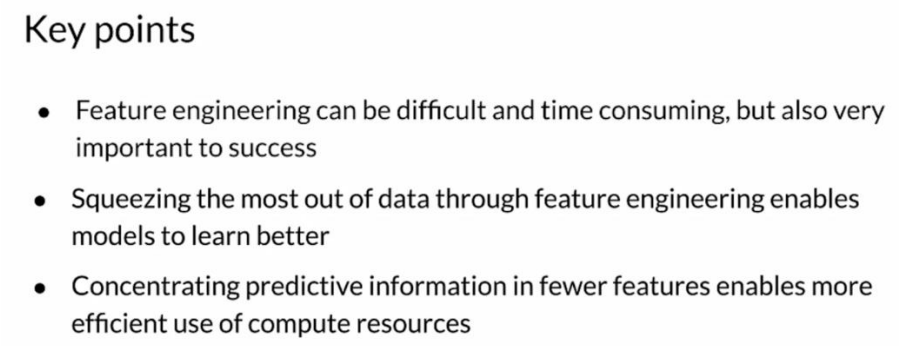


- The art of feature engineering tries to improve your model's ability to learn while reducing the compute resources it requires

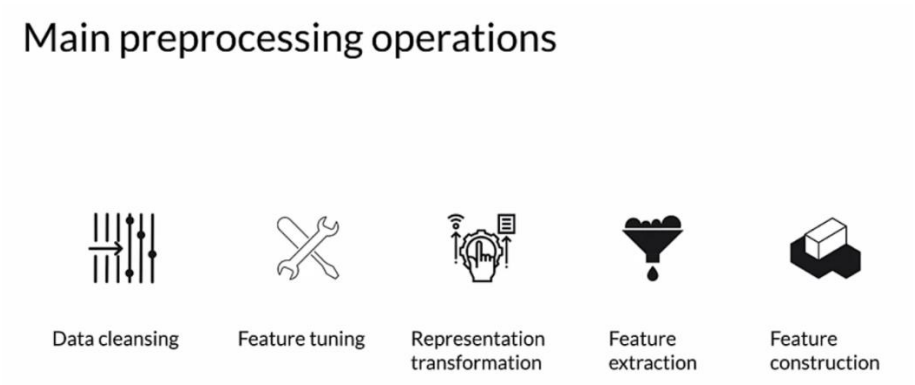
- It does this by transforming and projecting, eliminating and/or combining the features in your raw data to form a new version of your data set.
- Typically across the ML pipeline, you incorporate the original features often transformed or projected to a new space and/or combinations of your features.
- Objective function must be properly tuned to make sure your model is heading in the right direction and that it is consistent with your feature engineering.
- You can also update your model by adding new features from the set of data that is available to you
- Unlike many things in ML, this tends to be an iterative process that gradually improves your results as you iterate. You have to monitor that, and if it's not improving, maybe back up and take another approach.



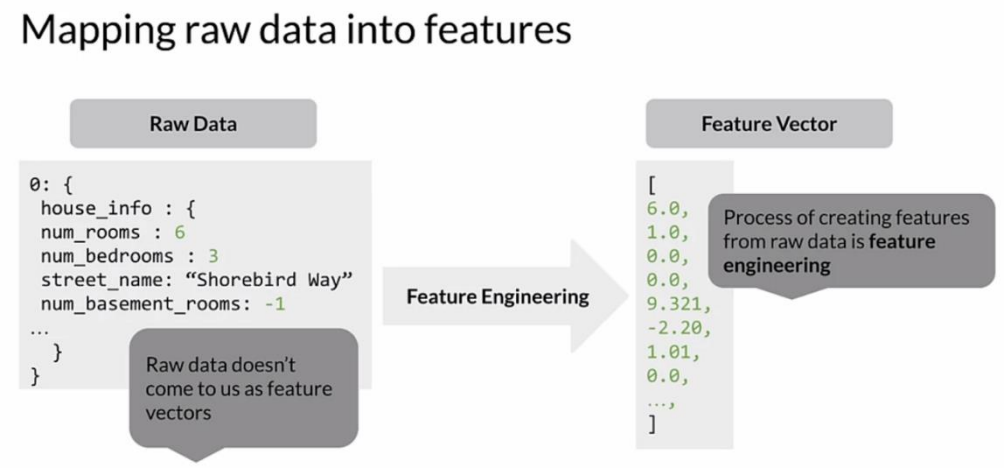
- Feature engineering is usually applied in two fairly different ways
- During training, you usually have the entire data set available to you. So you can use global properties of individual features in your feature engineering transformations.
- For example, you can compute the standard deviation of a feature and then use that to perform normalization.
- When you serve your trained model, you must do the same feature engineering so that you give your model the same types of data that it was trained on. For example, if you created a one hot vector for a categorical feature when you trained, you need to also create an equivalent one hot vector when you serve your model.
- However, during training and serving, you typically process each request individually, so it's important that you include global properties of your features, such as the standard deviation.
- If you use it during training, include that with the feature engineering that you do when serving. Failing to do that right is a very common source of problems in production systems, and often these errors are difficult to find.



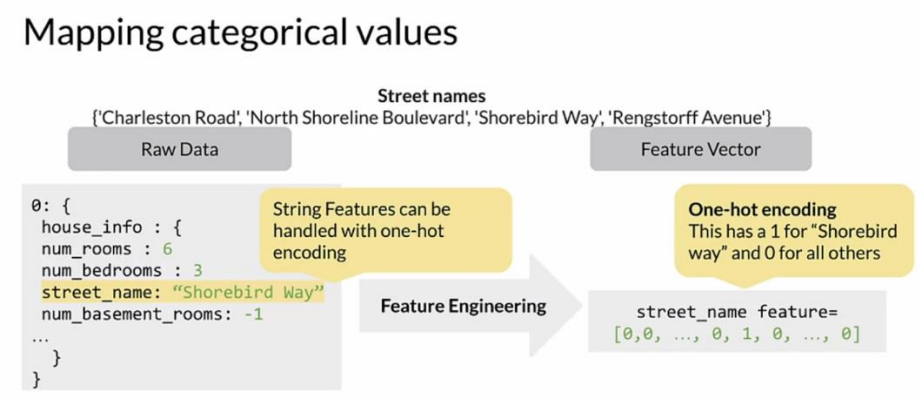
Preprocessing Operations



- First, let's go over what we're going to be talking about. We're going to be going into the main Preprocessing Operations. Obviously this is not going to be an exhaustive list. There's a lot of things you can do to data, but we'll cover some of the main ones anyway.
- Here's some of the main preprocessing operations. One of the most important Preprocessing Operations is Data cleansing, which in broad terms consists in eliminating or correcting erroneous data.
- You'll often need to perform transformations on your data, such as scaling or normalizing your numeric values
- Since models, especially neural networks, are sensitive to the amplitude or range of numerical features, data preprocessing helps Machine Learning build better predictive Models.
- Dimensionality reduction involves reducing the number of features by creating lower dimension and more robust data representation.
- Feature construction can be used to create new features by using several different techniques, which we'll talk about some of them.



- This is an example of data that we have. This is data from a house, but this is only what we start with. The raw data.
- Feature Engineering transforms the raw data and creates a feature vector from it.
- For example, integer data can be mapped to floats, numerical data can be normalized, and one-hot vectors can be created from categorical values.
- Feature Engineering creates features from raw data and you're probably already familiar with this

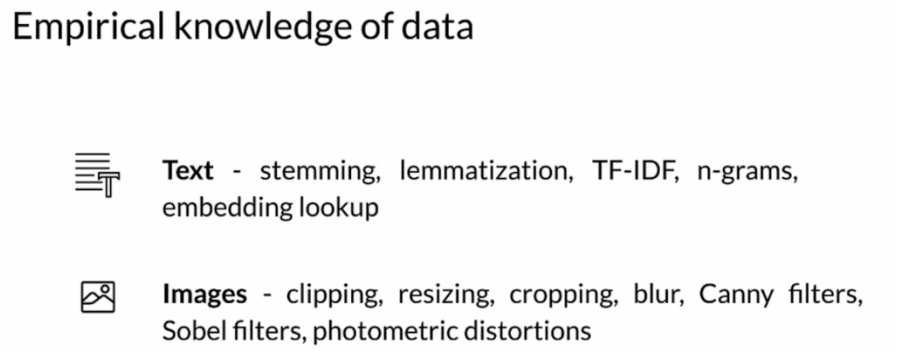


- In this example, we're going to take a categorical feature called Street name.
- We're going to one-hot encode the string feature as a way to make it better for a Model to learn from.

Categorical Vocabulary

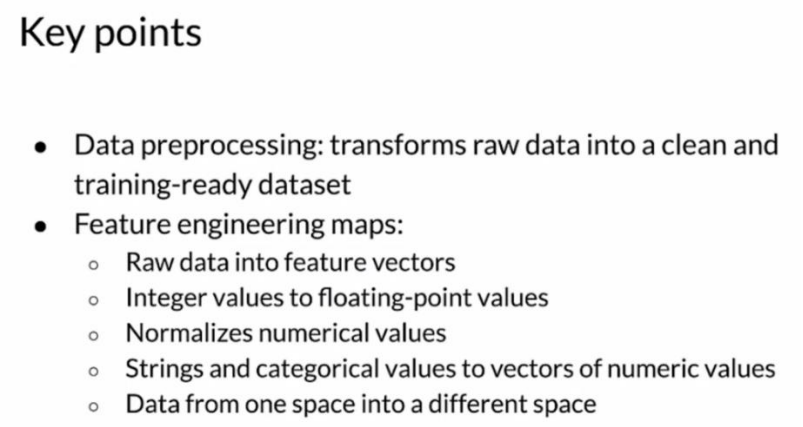
```
# From a vocabulary list

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_list(

                          key=feature_name,

                          vocabulary_list=["kitchenware", "electronics", "sports"])
# From a vocabulary file

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_file(

                           key=feature_name,

                           vocabulary_file="product_class.txt",

                           vocabulary_size=3)
```

- Creating a vocabulary is another way to do that. There are a couple of different ways to do it.
- Tensorflow provides different functions for creating columns of categorical vocabulary, and other frameworks do very similar things.
- Categorical column with vocabulary lists maps each string to an integer based on an explicit vocabulary lists. Feature name is a string which corresponds to the categorical feature and vocabulary list is an ordered list that defines vocabulary.
- Feature or categorical column with vocabulary file is used when you have two long list and this function allows you to put the words in a separate file.
- In this case, vocabulary list is defined as a file that will you define the list of words.

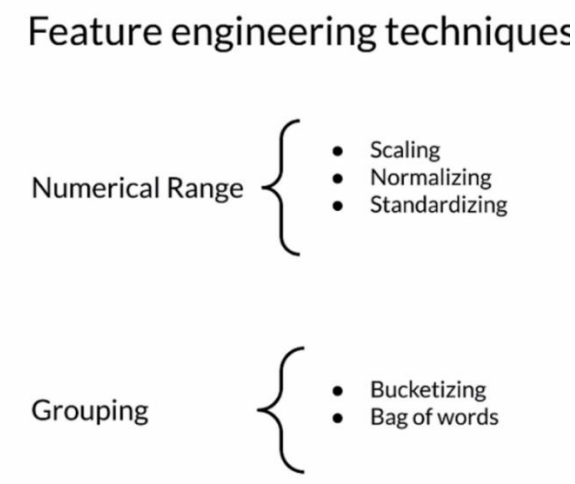


- But you may also know some things about your data, and part of that might be domain knowledge of the domain you're working in, or just knowledge of how to work with different data.

- There's very different operations and preprocessing techniques that can help you increase the predictive information in say, text data.
- For text, we have things like stemming and lemmatization and normalization techniques like TF-IDF and n-grams, embeddings, all of which focuses on the semantic value of the words.
- Images are similar. There's things that we will know about how we can improve the predictive qualities of images. Things like clipping them, resizing them, cropping them, blurring them
- We also often use filters like Canny filters or Sobel filters or other photometric distortions. All these things can really help us work with image data to improve its predictive quality.



Feature Engineering Techniques



- Feature Engineering takes a variety of forms, normalizing, and scaling features, and coding categorical values and so forth.
- This is going to be very dependent on the particular algorithm that you're going to use. You have to understand the connection between the kinds of scaling or grouping that you do, and the algorithms that are going to be using with it.

Scaling

- Converts values from their natural range into a prescribed range
  - E.g. Grayscale image pixel intensity scale is [0,255] usually rescaled to [-1,1]

```
image = (image - 127.5) / 127.5
```

- Benefits
  - Helps neural nets converge faster
  - Do away with NaN errors during training
  - For each feature, the model learns the right weights

- Scaling converts to standard range and different techniques do it differently.
- For example, a gray-scale image pixel intensity is typically expressed in the raw data as a number between 0-255, and then that's usually re-scaled to negative one to one.
- The benefits of that are, it helps the neural network to converge faster. It does away with a lot of the problems with NaN number errors during training.
- For each feature the model is really trying to learn the right weights, and having them in the right numerical range helps a lot.
- Scaling can be done via normalization or standardization

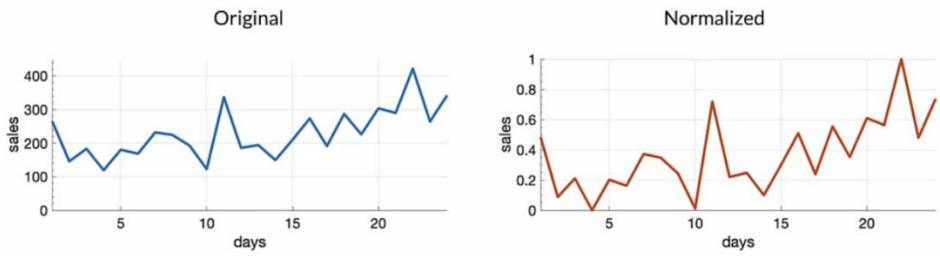
Normalization

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

$$X_{\text{norm}} \in [0, 1]$$

- Here's the standard formula for normalizing something. It's also called min-max.
- You have to make a full pass over your data set to get that minimum value, and then you subtract that and then you divide by the maximum value minus the minimum value.
- Then it gives you a number that is between 0-1.
- **Normalizations are usually good if you know that the distribution of your data is not Gaussian** (i.e. not a normal distribution). Doesn't always have to be true, but typically that's a good assumption to start with.

Normalization



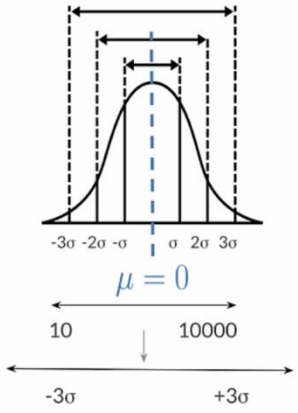
- Need to remember that you will need to make a full pass across the dataset in order to get the numbers you need to perform normalization

Standardization (z-score)

- Z-score relates the number of standard deviations away from the mean
- Example:

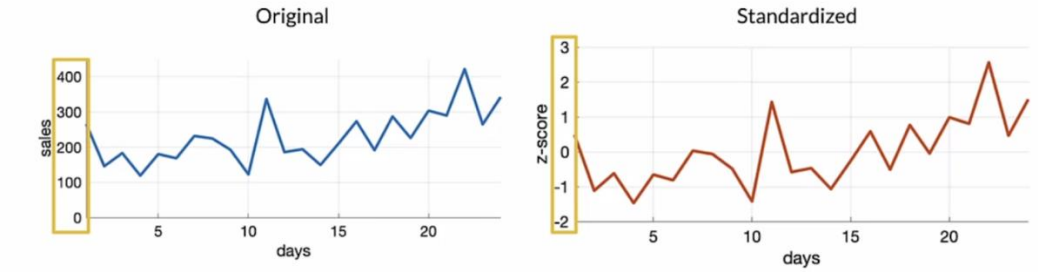
$$X_{\text{std}} = \frac{X - \mu}{\sigma} \quad \text{(z-score)}$$

$$X_{\text{std}} \sim \mathcal{N}(0, \sigma)$$

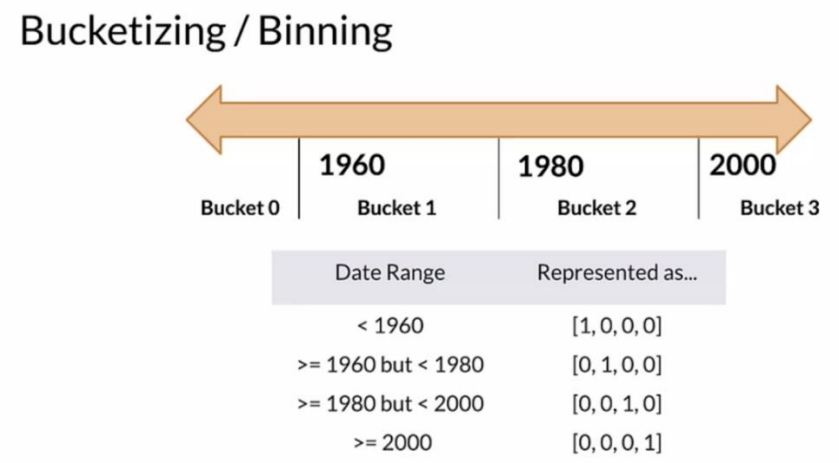


- Standardization, which is often using a Z-score, is a different technique.
- It's a way of scaling using the standard deviation. It's looking at the distribution itself and trying to scale relative to that.
- In the example here, we are going to subtract the mean and divide by the standard deviation. That gives you the Z-score or the standardized value of X, which is somewhere between zero and the standard deviation. This is how that's expressed, actually between some multiple of the standard deviation. But it's centered around the mean of the data.

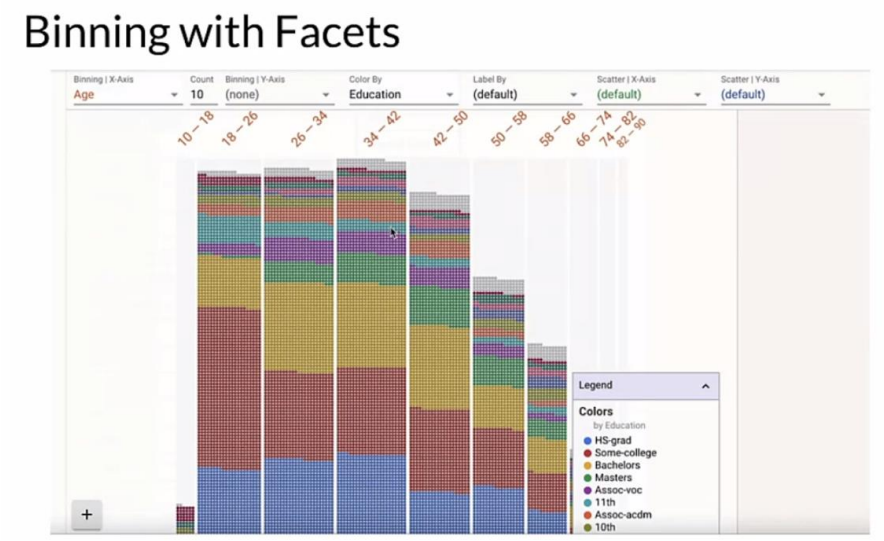
Standardization (z-score)



- Notice that this score is centered on zero, so the mean is translated to zero. But you can have negative values and positive values that are beyond one.
- If your data is a normal distribution, then a standardization technique is a good place to start
- But I encourage you to try both standardization and normalization and compare the results. Because sometimes it doesn't make much of a difference. Sometimes it can make a substantial difference. It's good to try both



- Moving on into bucketizing and binning. We're going to take a look at an example where we have a house that was built in a particular year and we're going to bucketize that.
- Often you don't want to enter a number directly into a model. Instead, you want to encode it as a category by grouping it into buckets. You notice how we've taken our number, our year, and we've created a one-hot encoding of that that helps us learn from that number in a more really relevant way because the difference between 1960 and 61 isn't nearly as important as the difference between 1960 and 1970 in terms of the value that this feature contained.
- Looking at how it gets binned into one-hot encoded vector, you can see it's put into different buckets.



- We can also look at that using Facets, which is a nice visualization tool to help us look at that.
- This really demonstrates some of the value of bucketizing and binning, but it also is a good way to look at your data and try to understand it.
- An important part anytime you're working with data, is making sure that you really have a good solid understanding of what's happening in your data, and develop an intuitive sense for it. This kind of visualization can really help you do that.

Other techniques

Dimensionality reduction in embeddings

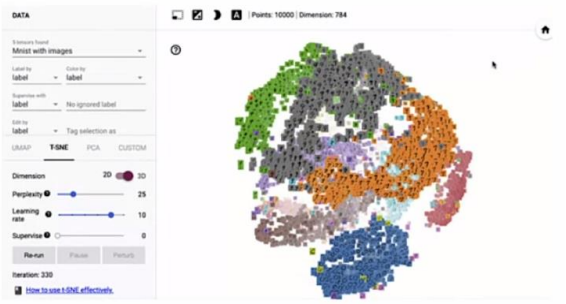
- Principal component analysis (PCA)
- t-Distributed stochastic neighbor embedding (t-SNE)
- Uniform manifold approximation and projection (UMAP)

Feature crossing

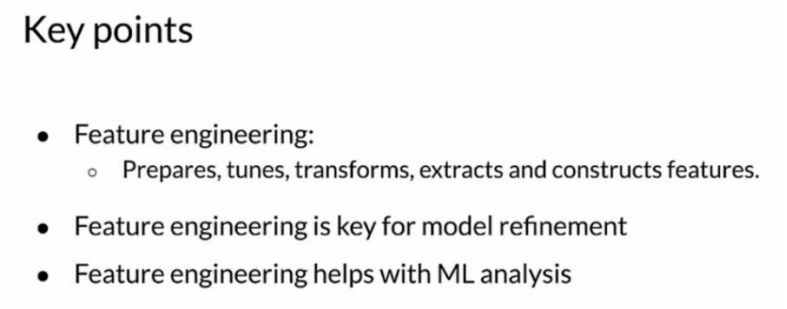
- Some other techniques. There's dimensionality reduction techniques, for example, that you can do. There's PCA, which is going to project your data along the principal components in order to reduce the dimensionality of it.
- There's t-SNE, which is an important technique for embeddings, often very useful.
- Uniform manifold approximation and projection is a less well-known technique, but has some interesting aspects. We won't really go into that in detail. Then feature crossing as well.

TensorFlow embedding projector

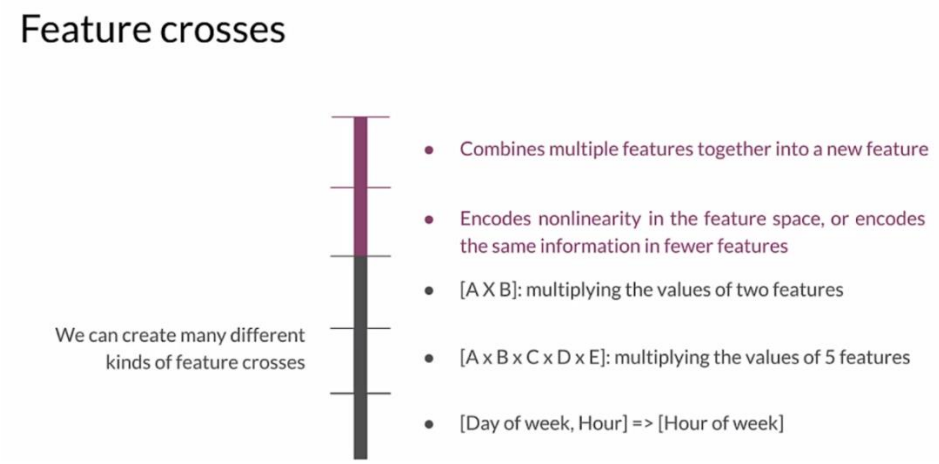
- Intuitive exploration of high-dimensional data
- Visualize & analyze
- Techniques
  - PCA
  - t-SNE
  - UMAP
  - Custom linear projections
- Ready to play



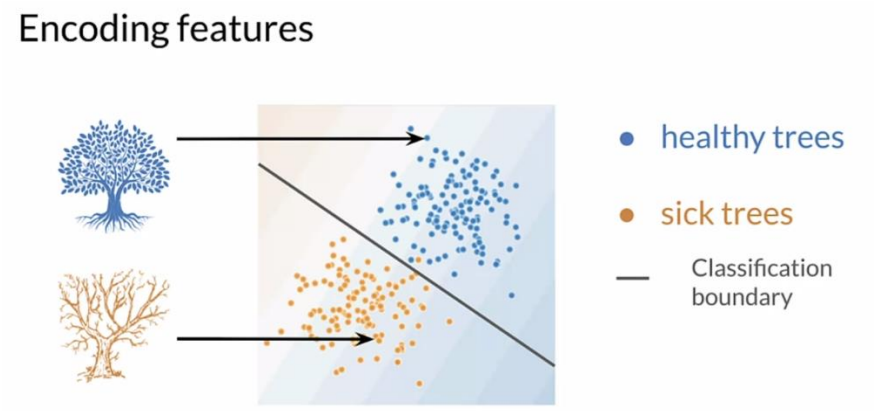
- One of the things that you can use when you're working with embeddings is an embedding projector, like the TensorFlow embedding projector that we're looking at here.
- You can quickly get an idea of what your data looks like at a particular space. That again, is helping you to develop that intuitive sense of your data. You can already see just looking at it, you can get a feel for how it's clustered. You can see where different types of data land in that space.
- This is useful as an intuitive exploration, really important for high-dimensional data because we as humans can visualize maybe three dimensions before it gets really weird.
- TensorFlow embedding projector is available. It's free, you can go play with it. It's actually a lot of fun, but it's also a great tool to really help you understand your data.
- Link to Tensorflow embedding projector: https://projector.tensorflow.org/



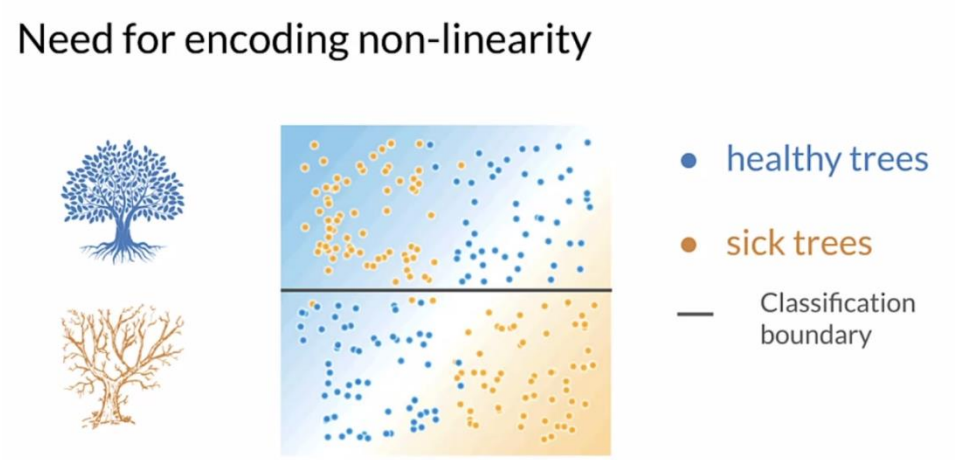
Feature Crosses



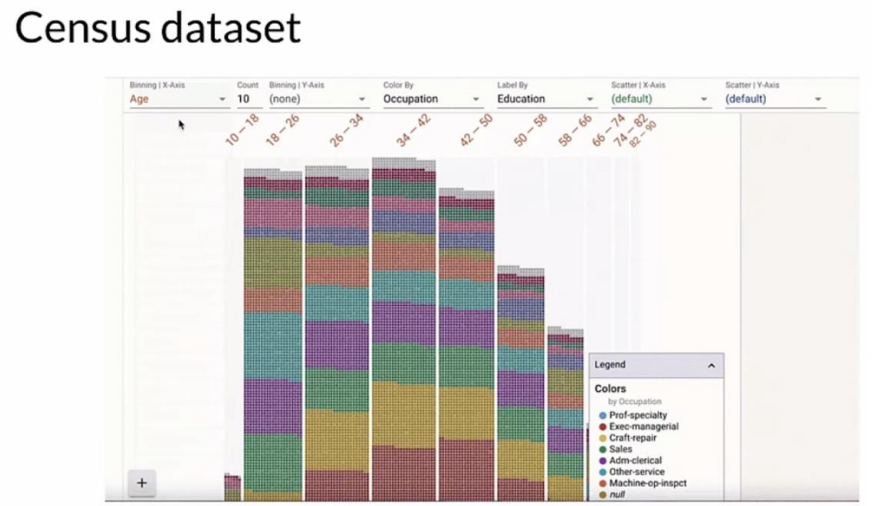
- What are Feature crosses? Well, they combine multiple features together into a new feature. That's fundamentally what a feature across.
- It encodes non-linearity in the feature space, or encodes the same information with fewer features.
- We can create many different kinds of feature crosses and it really depends on our data.
- It requires a little bit of imagination to look for ways to try to do that, to combine the features that we have.
- For example, if we have numerical features, we could multiply two features [A x B] and produce one feature that has an expression of those two features.
- If we have 5 features, we could multiply them all together if there are numerical features and then end up with one feature instead of say five.
- We can also take categorical features or even numerical features and combine them in ways that make a semantic sense.
- Taking the day of the week and hour, if those are two different features that we have, putting those together, we can express that as the hour of the week and we have the same information in one feature.



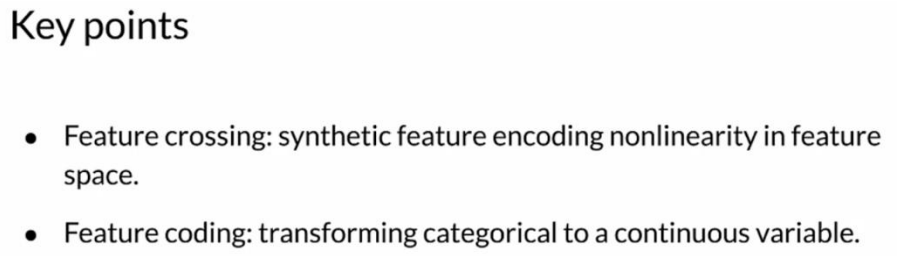
- To encode features, let's look at some examples. Suppose we have a data-set that's looking at healthy trees and sick tree. We're going to use a scatter-plot to try to understand our data.
- We're looking at the scatter plot and we use some visualization tools, and we ask ourselves, can we draw a line to separate these two groups, these two clusters?
- If we can use a line to create the decision boundary, then we know that we can use a linear model.
- In this case, looking at this, we could use a line to do that separation. That's great.



- Linear models are very efficient. But suppose the data looks like this. Not so easy anymore.
- This becomes a non-linear program. Then the question is, can we project this data into a linear space and use it with a linear model, or do we have to use a non-linear model to work with this data?
- Because if we try to draw just a linear classification boundary with a data as is, it doesn't really work great.

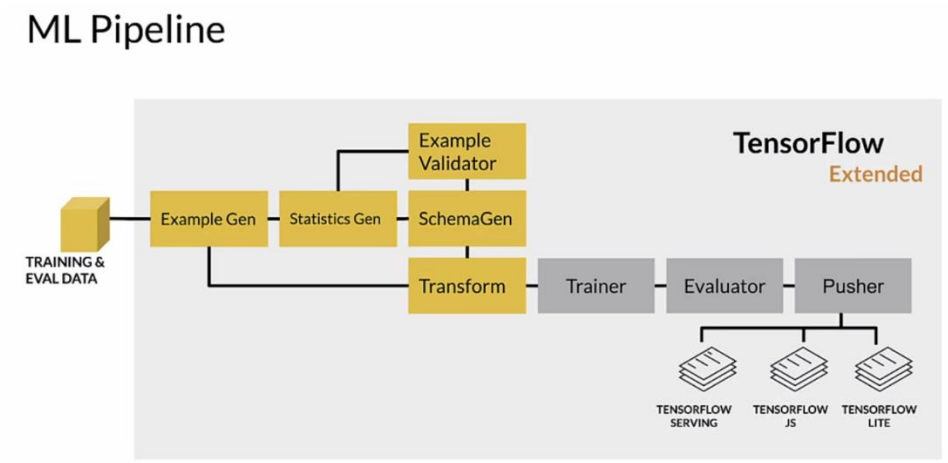


- Again, we can use visualization tools to help us understand this.
- Looking at our data really helps inform us and guide the choices that we make as developers about what kind of models we choose and what feature engineering we apply.



Preprocessing Data at Scale

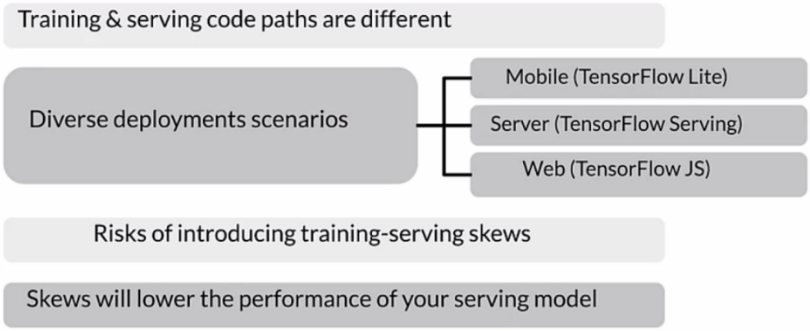
- We've probably all done feature engineering before, but it's one thing to do it in a once-off notebook with maybe a few 100 megabytes of data. And quite another thing to do it in a production environment with maybe a couple of terabytes of data in a repeatable, reproducible, consistent and automated process.
- Let's take a look now at how to do feature engineering at scale



- It is important to use a pipeline, a unified framework where you can both train and deploy with consistent and reproducible results

- To preprocess data at scale, we start with real world models and these could be terabytes of data.
- So when you're doing this kind of work, you want to first start with a subset of your data.
- Large scale data processing frameworks are no different than what you're going to use, on your laptop or in a notebook or what have you. So you need to start thinking about that from the beginning of the process, how this is going to be applied
- The earlier you can start working with those frameworks, the more you can work out the issues early with smaller datasets and done so with quicker turnaround time
- Consistent transformations between training and serving are incredibly important. If you do different transformations when you're serving your model than you did when you were training, your model is going to have problems, and often those problems are very hard to find.

Inconsistencies in feature engineering



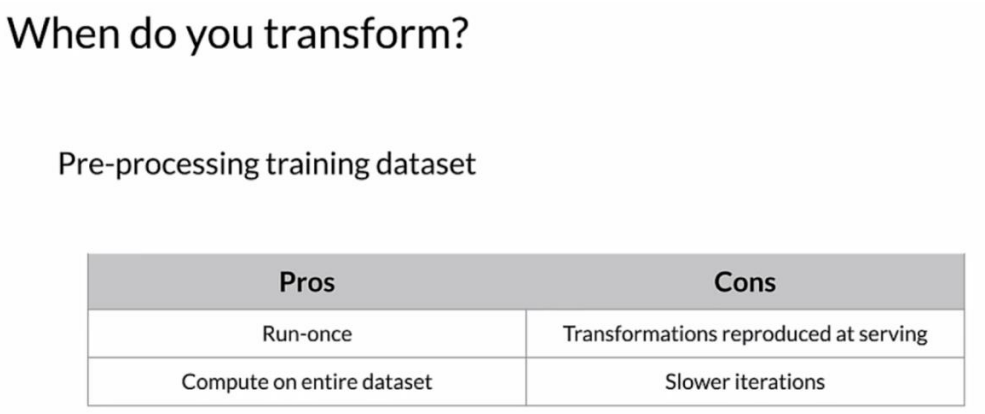
- When you are training your model, you have code that you're using for training.
- If you're using different code for training (e.g. Python) but different code for serving (Java), that's going to be a potential source of problems.
- Another cause of inconsistencies is that you could have different deployment scenarios, where you might be deploying your model for in different environments.
- You could be using the same model in a servant cluster and also using it on an IOT device as well.
- So there's different deployment targets and you need to think about those and what are the CPU resources or the compute resources that you have available on those targets.
- Mobile, for example, have very tight compute resources, whereas for servers, you have more luxury. But again, cost is a big factor and in a web browser, that could be deployed on different clients.
- There is risks in introducing training, serving skews especially when we have different code paths between training and serving i.e. don't have exactly the same transformation is happening between the two
- The best way to do that is to use exactly the same code.

Preprocessing granularity

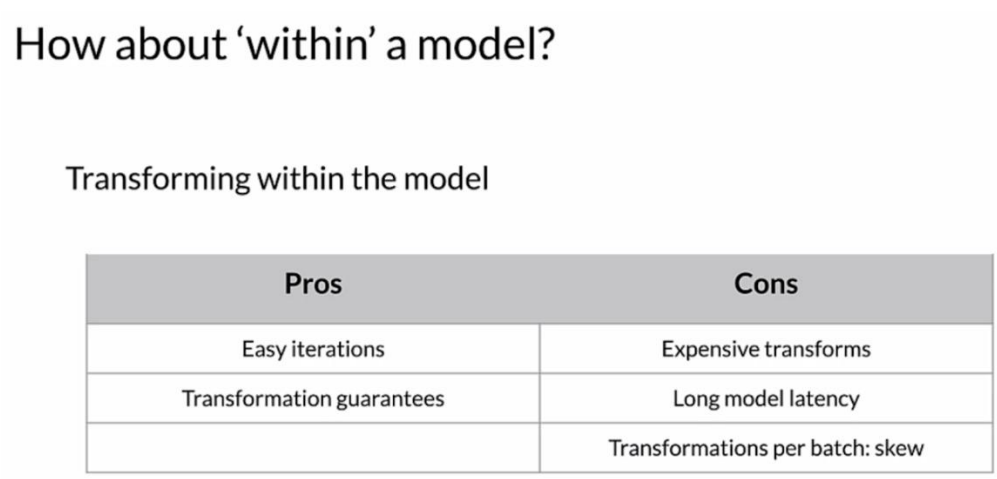
Transformations	
Instance-level	**Full-pass**
Clipping	Minimax
Multiplying	Standard scaling
Expanding features	Bucketizing
etc.	etc.

- There is a notion of the granularity when you're doing preprocessing.
- You're going to do transformations, both at the instance level and as a full pass over your data. Depending on the transformation that you're doing, you may be doing it in one or the other.
- Full pass requires that you have the whole dataset. For example, for clipping, even for clipping, you need to set clipping boundaries. For instance level, you can set some arbitrary setting of those boundaries
- Feature cross is an example of instance level transformation
- But if you're using the data set itself to determine how to clip, then you're going to need to do a full pass.
- Doing a multiplication at the instance level is fine, but standardization will need the standard deviation, which means full pass is needed.
- Unless I know ahead of time what buckets are going to be, I'm going to need to do a full pass to find what buckets makes sense for bucketizing.

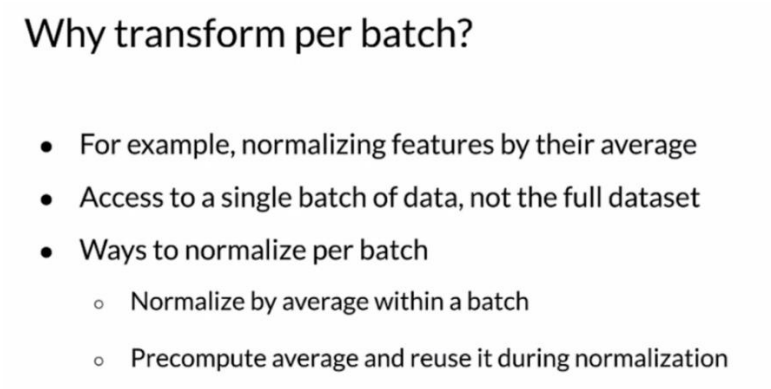
- Anything I need to do that requires characteristics of the whole dataset, I need to have that saved off so I can use it at serving time.



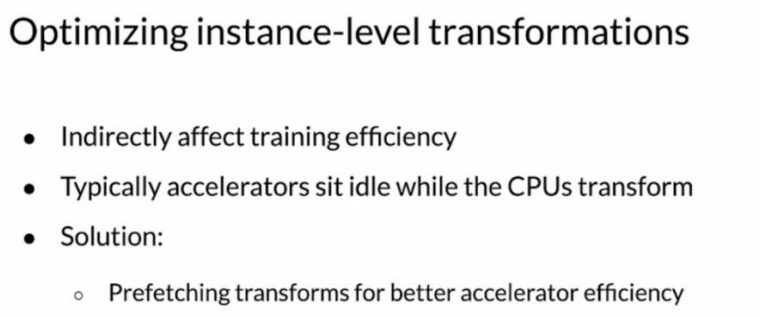
- So when do you transform? You can pre-process your training dataset, and there's pros and cons in how you do that.
- First of all, an advantage is that you only run the pre-processing once per training session. Furthermore, it is done over the entre dataset.
- The cons are that you will need to reproduce all those transformations when serving, or save off the information that you learned about the data, like the standard deviation.
- Another issue is the slower iterations around this. Each time you make a change, you've got to make a full pass over the dataset.



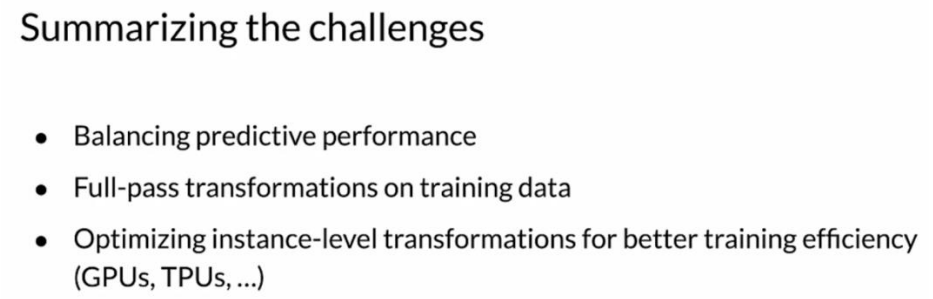
- So you can do pre-processing within the model.
- First of all, it makes iteration somewhat easier (and faster) because pre-processing is embedded as part of your model, and there's guarantees around the transformations that you're doing.
- But the cons are it can be expensive to do those transforms, especially when your compute resources are limited.
- You may have say GPUs or TPUs when you're training, but you may not have those resources when you're serving. So how models get applied during serving is a key consideration, as feature engineering within the model increases inference time.
- There is also the issue of long model latency, that's when you're serving your model.
- If you're doing a lot of transformation within the mode, it can slow down the response time for your model and increase latency.
- Feature engineering within the model is also limited to batch computations



- You can also transform per batch instead of for the entire dataset.
- So you could for example, normalize features by their average within the batch. That only requires you to make a pass over each batch and not the full data set, especially if you're working with terabytes of data. That can be a significant advantage in terms of processing.
- And there are different ways to normalize per batch as well. You can compute an average for normalization per batch, and then do it again for the next batch. There will be differences from batch to batch.
- You can also precompute the average and use it during normalization, such that you can use it for multiple batches



- You need to think about optimizing the instance level transformations as well, because this is going to affect both the training efficiency and your serving efficiency.
- Your accelerators (e.g. GPU, TPU) may be sitting idle while your CPU is doing the transforms. That's something that you want to try to avoid because accelerators tend to be expensive.
- As a solution, you can prefetch your transformations and use your accelerators more efficiently. So by prefetching, you can prefetch with your CPU, feed your your accelerator, and parallelize that processing.
- This all boils down to cost, how much it costs to train, time required to train, and how efficient it is.



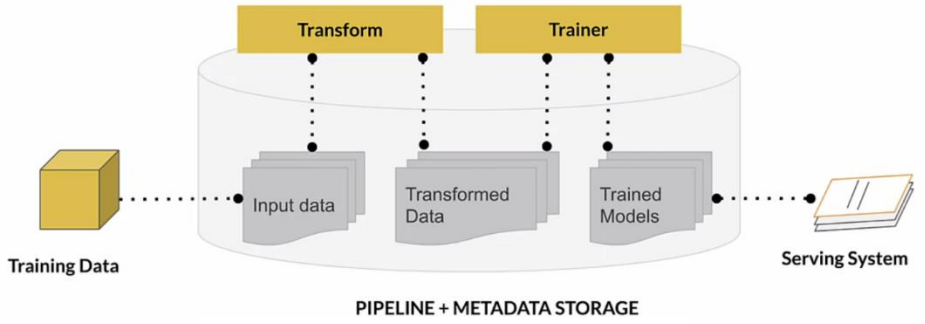
- So to summarize the challenges, we have to balance the predictive performance of our model and the requirements for it.
- Passing transformations on the training data is one thing. If we're going to do that, then we need to think about how that works when we serve our model, as well as saving those constants.
- And we want to optimize the instance level transformations for better training efficiency. So things like prefetching can really help with that.

Key points

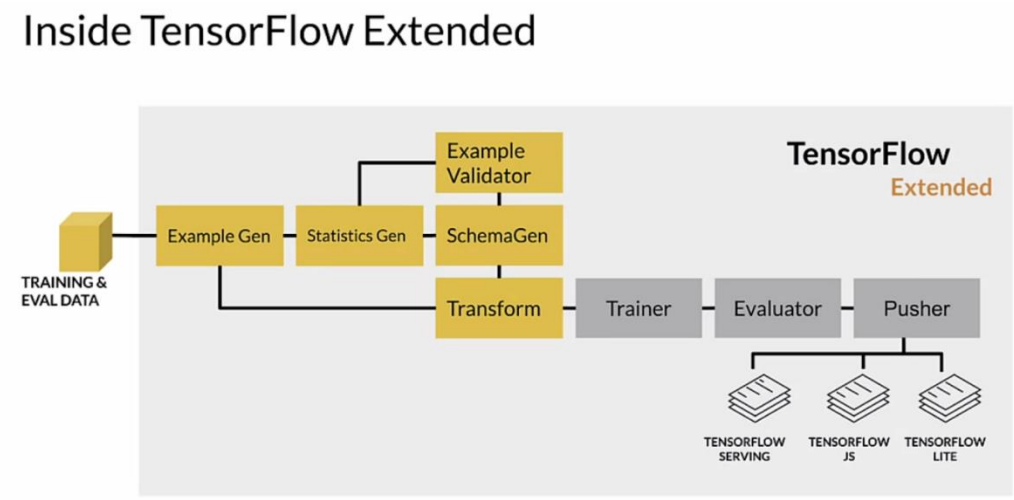
- Inconsistent data affects the accuracy of the results
- Need for scaled data processing frameworks to process large datasets in an efficient and distributed manner

Tensorflow Transform

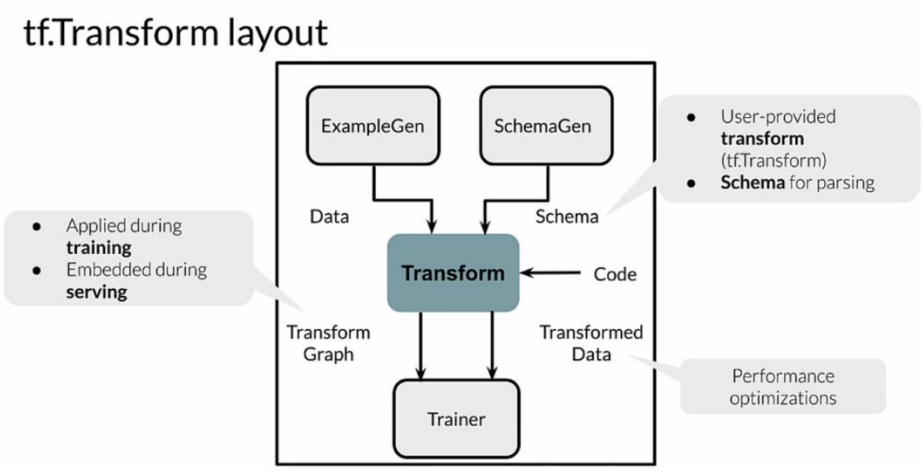
Enter tf.Transform



- We are going to look at the benefits of using TensorFlow Transform and how it applies feature transformations, and we are going to look at some of the analyzers, and the roles they play in doing pre-processing.
- Let's talk about transform. First off, it does what you might think it would do, it takes Training Data, and it's going to process it.
- The end result is going to be deployed into the Serving System.
- In the middle, there's a PIPELINE that is used, and there is METADATA that forms a key role in organizing and managing the artifacts that are produced as data is transformed.
- One of the reasons that's important, is because, we want to understand the lineage or provenance of those artifacts, and be able to connect them, and find the chain of operations that generated each of them.
- We have our Training Data, that is the input data into the system that forms an artifact, we give that to Transform and it transforms our data. It's going to take our raw data and move it into the form that we're actually going to use at our feature engineering.
- The transformed data is given to the Trainer which is going to do training. Transform and Trainer here are both components in a ML pipeline, specifically a TFX ML pipeline. The output from the Trainer is of course a Trained Model, that is also an artifact.
- That gets delivered to the serving system or whatever system we're using, where it's going to be used to run inference.

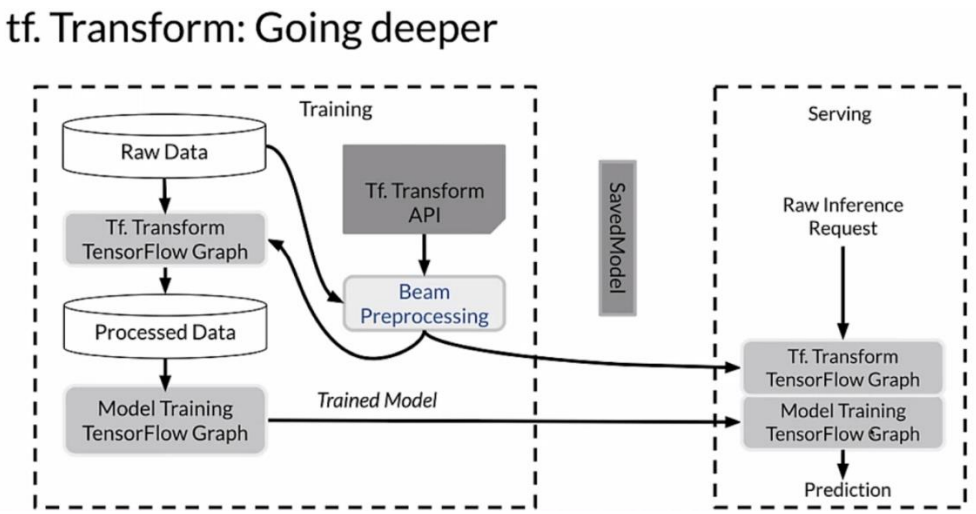


- Looking at this a little differently, we're starting with our training and eval data.
- ExampleGen does that train/eval split, and the split data gets fed to StatisticsGen.
- Note that these are both TFX components within a TFX pipeline, so ExampleGen is a component, and StatisticsGen is a component.
- StatisticsGen calculates statistics for our data. For each of the features, if they're numeric features, for example, what is the mean of that feature? What is the standard deviation? The min, the max, so forth.
- Those statistics get fed to SchemaGen, which infers the data types of each of our features. That creates a schema that is then used by downstream components including Example Validator, which takes those statistics and that schema, and looks for problems in our data.
- We won't go into great detail here about the things that Example Validator looks for, but it looks for things where we have the wrong type in a particular feature (e.g. an integer where we expected a float).
- Now, we're getting into transform. Transform is going to take the schema that was generated on the original split dataset, and it's going to do our feature engineering.
- Transform is where the feature engineering happens.
- The transformed output then is given to the Trainer, there's Evaluator that evaluates the results, a set of Pusher that pushes the model to our deployment targets where we serve our model e.g. TENSORFLOW SERVING, or JS, or LITE.

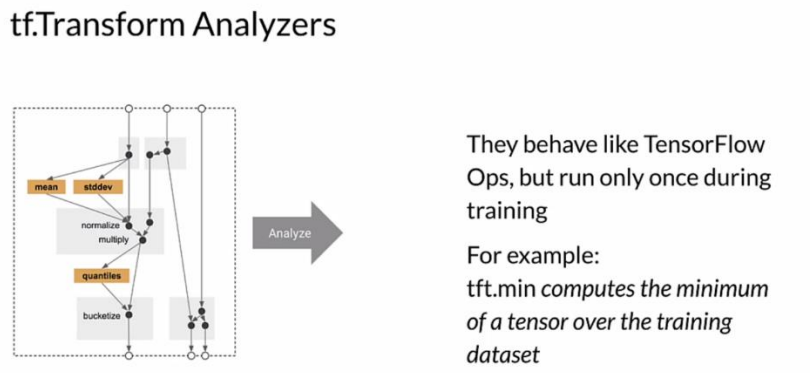


- Internally, if we want to look at the transform component, it's getting inputs from ExampleGen and SchemaGen. That is the data that was originally split by Example Gen, and the schema that was generated by SchemaGen.
- That schema, by the way, may very well have been reviewed and improved by a developer who knew more about what to expect from the data. That's referred to as **curating** the schema.

- Transform also receives a lot of user code because we need to express the feature engineering we want to do. If we're going to normalize a feature, we need to give it user code to tell transform to do that.
- The result is a **TensorFlow graph**, which is referred to as the **transform graph** and the transform data itself.
- The graph expresses all of the transformations that we are doing on our data.
- The transform data is simply the result of doing all those transformations. Those are given to trainer, which is going to use the transformed data for training and it's going to include the transform graph
- So far we have a user provided transform component and we have a schema for it. We apply our transformations during training and also apply those transformations at serving time.

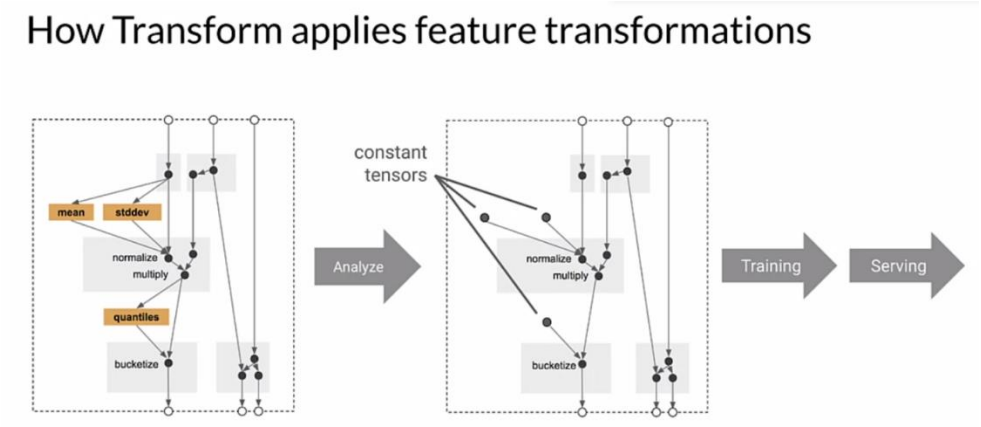


- Continuing a little deeper here, we have our raw data and we have our transform component. The transform component. The result of running that graph is processed data, which has been transformed.
- The transformed data is then passed into the Trainer component for model training. Training our model creates another TensorFlow graph.
- Notice now we have **two different graphs**. We have a graph from transform, and a graph from training.
- Using the Tf.Transform API, we express the feature engineering that we want to do. The transform component gives that pre-processing code to a **Apache Beam** distributed processing cluster.
- Remember that our pipeline is designed to work with potentially terabytes of data. That could be quite a bit of processing to do, so we make use of a distributed processing cluster with Apache Beam, which gives us the capacity to do that.
- The result is a saved model.
- We then pass both of those graphs and the saved model into our serving infrastructure. These components are then used to serve requests to obtain predictions

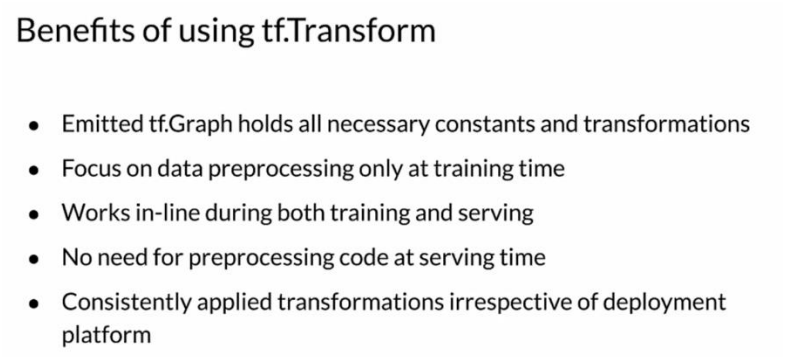


- Looking at this a little differently, we have analyzers.
- We're using TensorFlow Ops and a big part of what an analyzer does is that it makes a full pass over our dataset in order to collect constants that we're going to need when we do feature engineering

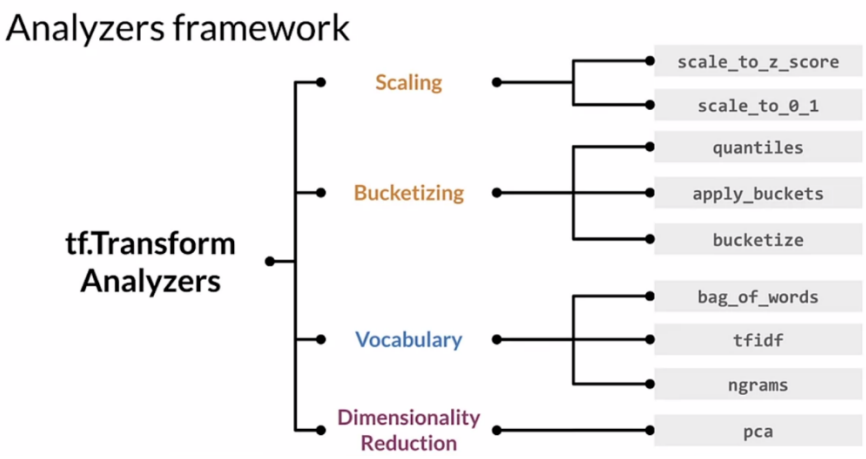
- For example, if we're going to do a min-max, we need to make a full pass through our data to know what the min and the max are for each feature. Those are constants that we need to express.
- We're going to use an analyzer to make that pass over the data and collect those constants. It's also going to express the operations that we're going to do.
- They behave like TensorFlow Ops, but they only run once during training, and then they're saved off as a graph. For example, if we're using the tft.min, which is one of the methods in the transform STK, it'll compute the minimum of a tensor over the training dataset.



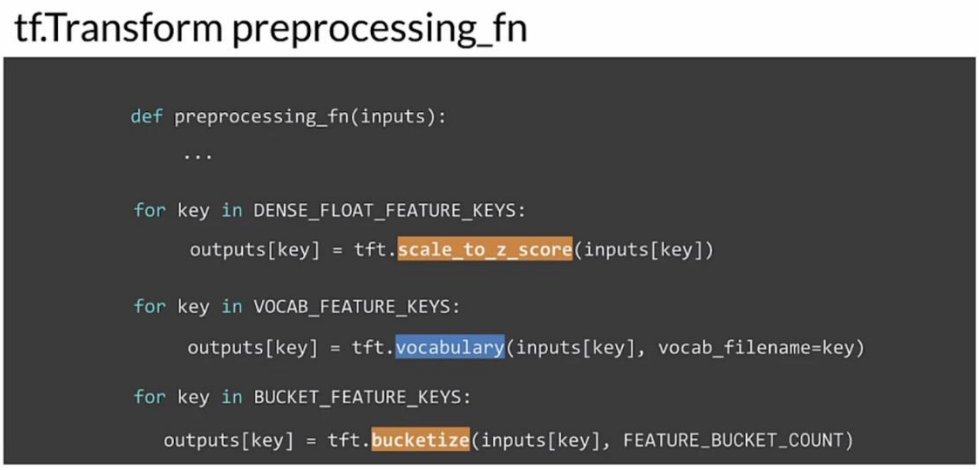
- Let's look at how this gets applied.
- We have the graph here, that is our feature engineering. We run an analysis across our dataset to collect the constants that we need. What we're really doing is we apply these into our Transform graph, so that we can transform individual examples without making a pass over our entire dataset again later.
- The same exact graph obtained from training gets applied during serving. Hence, there will be no potential here for training and serving skew, or having different code paths when we train our model versus when we serve our model.
- We're using exactly the same graph in both places, so there is no potential for problems resulting from them not being equivalent
- This means that tf.Transform **eliminates the risk of introducing training-serving skew**



- What is the benefits of that? Well, the emitted graph that Transform emits is it has all necessary constants and transformations that we're going to need.
- The focus is on data pre-processing only at training time. It's doing that processing as well as generating that graph.
- It works in-line during both training and serving as we prepend it to our trained model, so there's really no need for pre-processing code at serving time.
- It consistently applies those transformations irrespective of the deployment platform. Remember, we could be deploying our model to a TensorFlow server or a mobile device using TensorFlow Lite or a web browser using TensorFlow Jazz. By using the same TensorFlow graph in all places, we're going to get exactly the same transformations.



- The Analyzers framework itself has different aspects.
- First of all, you can do scaling. Things like scaling with a z-score or just scaling between zero and one
- You can do bucketizing, where we can bucketize the feature into quantiles, for example. We're going to set up a set of buckets and then we're going to take those values that we have and assign them to the bucket so that we have categorical values for numerical ranges of value
- We can also do bocabularies, with things like tf-idf, bag of words or n-grams
- You could do dimensionality reduction too. You can do a PCA Transform to reduce the dimensionality of your data.



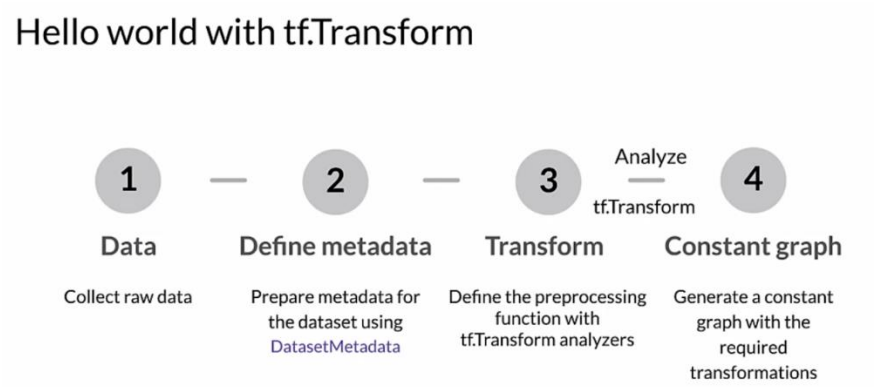
- Now let's look at some code. We're going to create a pre-processing function. This is the entry point that we're going to use to define the user code that expresses the feature engineering they're going to do.
- For example, we may make a pass over our data to look for floats. For our floats, we're going to scale those using a z-score.
- A vocabulary is also very similar. If we have vocabulary features, we would do this thing. These constants here are lists of the keys for each of the features that we want for particular transformations.
- Bucketing features, exactly the same thing. That bucket feature key is constant, is just a list of the keys for the features that we want to bucketize

Commonly used imports

```
import tensorflow as tf
import apache_beam as beam
import apache_beam.io.iobase

import tensorflow_transform as tft
import tensorflow_transform.beam as tft_beam
```

- You're going to need to import TensorFlow, obviously.
- Apache Beam as well, because remember, **Transform is going to use Apache Beam to distribute processing across a cluster**.
- Now you can also just use Beam on a single system or you can just run it on your laptop. It has something called the **DirectRunner**,where you don't need to actually set up a cluster. You can just run it, and there's nothing to more to setup in terms of infrastructure.
- In development, that's pretty useful. In deployment, unless you're working with a small amount of data, you probably want more compute horsepower than that
- apache_beam.io.iobase is part of what was described above. It helps us with IO in different formats and different ways reading and writing.
- TensorFlow Transform itself, obviously. Often we use the abbreviation of TFT for that.
- And then Transform itself has a Beam part of its modules, and it's often nice to pull out separately and work with. Those are usually the basic imports that we need to work with Transform.

Hello World with tf.Transform



- Now let's put this all together and look at a Hello world example with tensorflow Transform.
- We're going to collect some raw data and we're going to define the metadata around that data.
- Then we're going to run transform to transform the raw data into features and produce a graph
- Then we're going to in the background, where we're going to be running analyzers and using tf.Transform

Collect raw samples (Data)

```
[
    {'x': 1, 'y': 1, 's': 'hello'},
    {'x': 2, 'y': 2, 's': 'world'},
    {'x': 3, 'y': 3, 's': 'hello'}
]
```

- So we start with collecting our raw data which in this case is very simple, it's just some static data that we've created for the example. So we have three different features here, x, y and s.

Inspect data and prepare metadata (Data)

```
from tensorflow_transform.tf_metadata import (
    dataset_metadata, dataset_schema)

raw_data_metadata = dataset_metadata.DatasetMetadata(
    dataset_schema.from_feature_spec({

    'y': tf.io.FixedLenFeature([], tf.float32),
        'x': tf.io.FixedLenFeature([], tf.float32),
        's': tf.io.FixedLenFeature([], tf.string)
    }))
```

- So we express the types of those features and information about them as metadata.
- We're going to import the metadata module with Tensorflow transform and we're going to express that metadata using a feature spec.
- We can create a feature spec that expresses some information about our our feature. So this is telling us that, y is a float feature first of all, and that it's a fixed length feature.
- So this is not an example of a sparse feature or ragged tensor, but rather a fixed length feature. So both y and x are fixed length float features, and s is a string feature with fixed length

Preprocessing data (Transform)

```
def preprocessing_fn(inputs):
"""Preprocess input columns into transformed columns."""
    x, y, s = inputs['x'], inputs['y'], inputs['s']
    x_centered = x - tft.mean(x)
    y_normalized = tft.scale_to_0_1(y)
    s_integerized = tft.compute_and_apply_vocabulary(s)
    x_centered_times_y_normalized = (x_centered * y_normalized)
```

- So now we go this is our preprocessing function. This is where our user code is going to go.
- It's the entry point for our user code, so you can see we're pulling the values for x, y and s from our inputs and we're going to do some transformations.
- These are very simple transformation. So we are going to center x and that's going to require the mean. So that is, in the background without us thinking about it, going to make a pass over the to get the mean.
- We're also going to do a normalization using very simple scaling between zero and one. That is only going to do one pass over the data
- Then we're going to take that string feature s, and we're going to create a vocabulary for it so that we get an integer value for s.
- And then we create a feature cross, so this is purely synthetic feature of x centered times y normalized.

Running the pipeline

```
def main():
with tft_beam.Context(temp_dir=tempfile.mkdtemp()):
  transformed_dataset, transform_fn = (
      (raw_data, raw_data_metadata) | tft_beam.AnalyzeAndTransformDataset(
          preprocessing_fn))
```

- So this is one of the ways that you can run transform. I just want to make you aware that this part that we're looking at here is not how it looks when you run it in TFX.
- So here we're just running it inside a main function and we're going to do that using Apache beam. That's going to require us to establish a context for Apache beam.
- And then we're going to define a beam pipeline using the beam python syntax, which is a little bit different than you might expect.
- One of the things to get used to here is the pipe operator. So what that says is I'm going to run tft beam (I've established that in the context), I'm going to use AnalyzeAndTransformDataset to run preprocessing
- That's going to return a tuple result. So it's going to return both the raw data and the raw data metadata. So this python syntax is a little bit different than what you might be used to for python, and that is specific to Apache Beam.

Running the pipeline

```
transformed_data, transformed_metadata = transformed_dataset

print('\nRaw data:\n{}\n'.format(pprint.pformat(raw_data)))
print('Transformed data:\n{}'.format(pprint.pformat(transformed_data)))

if __name__ == '__main__':
  main()
```

- So we have our transformed data, we have our transform metadata, and we get both of that from our transformed data set.
- Now we can print out the raw data and the transformed data, and run our main function

After transforming with tf.Transform

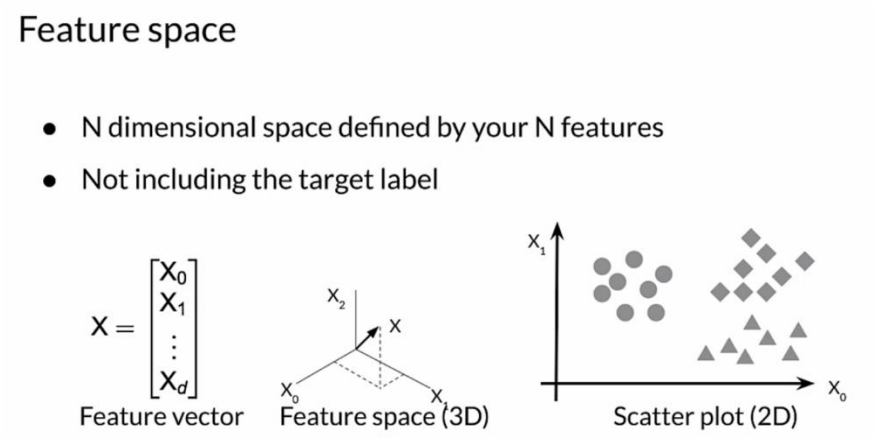
```
# After transform
[{'s_integerized': 0,
  'x_centered': -1.0,
  'x_centered_times_y_normalized': -0.0,
  'y_normalized': 0.0},
 {'s_integerized': 1,
  'x_centered': 0.0,
  'x_centered_times_y_normalized': 0.0,
  'y_normalized': 0.5},
 {'s_integerized': 0,
  'x_centered': 1.0,
  'x_centered_times_y_normalized': 1.0,
  'y_normalized': 1.0}]
```

- After running transform, we have features like this.
-

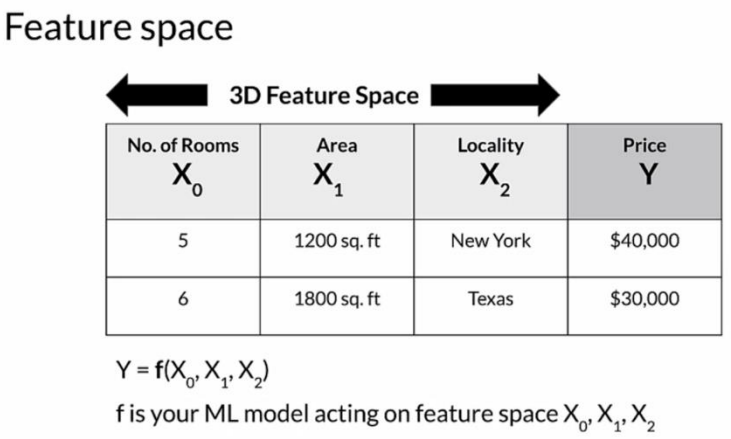
Key points

- tf.Transform allows the pre-processing of input data and creating features
- tf.Transform allows defining pre-processing pipelines and their execution using large-scale data processing frameworks
- In a TFX pipeline, the Transform component implements feature engineering using TensorFlow Transform

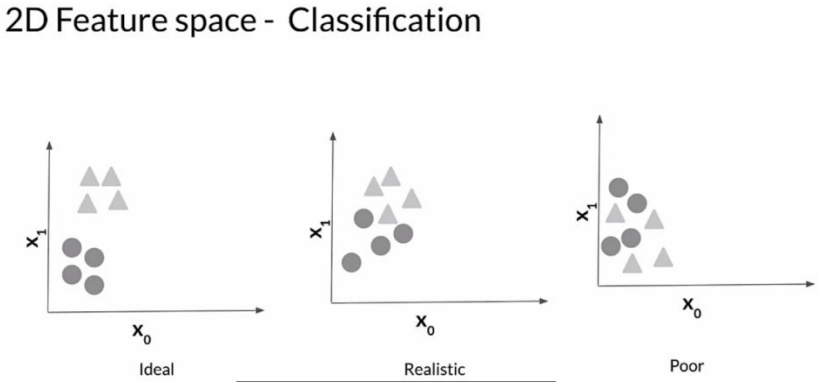
Feature Spaces



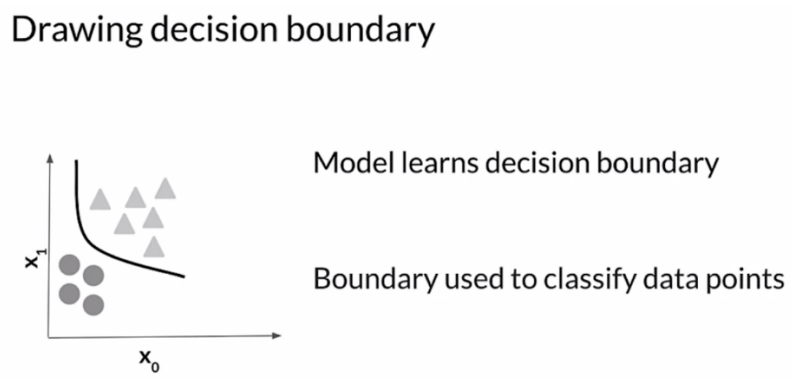
- A feature space is the n dimensional space that your features are defined.
- If you have two features, a feature space is two dimensional. If you have three features, its three dimensional and so forth.
- This does not include the target label, so that we're just talking about your features.
- If this is your feature factor, you have a feature vector X. And it has values from zero to D, which is the dimensionality of that vector.
- With a two dimensional feature vector, we could express it as a scatter plot in a 2D space. Those are fairly easy for us as humans to imagine.



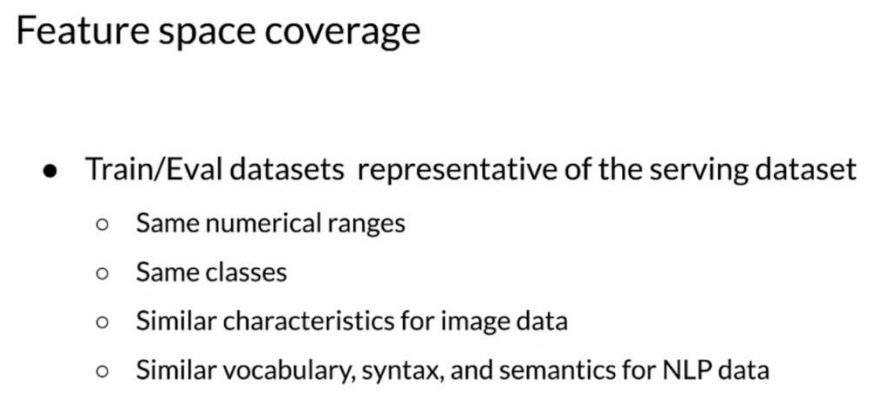
- Here's an example using features that you might have for a house, so the number of rooms for the house, the square footage, and where it's located at. So maybe we're trying to figure out the price of the unit.
- So in this case, **f** is your model and it's acting in your feature space, and it's going to produce a result Y



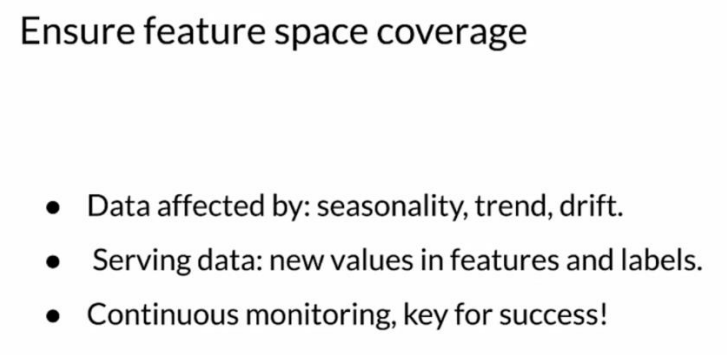
- Looking at a classification problem, where we have a two dimensional space. We have different distributions of the examples within our feature space.
- In an ideal case, they're easily separable, ideally with a linear function
- In a realistic case, maybe we can do that with a linear function if we sort of fudge around some of the some of the examples
- For a case with poor examples, it will be difficult to do separation with a linear function. Maybe we just have to use a non-linear model



- What we're going to try to do is draw a boundary within that feature space.
- The boundary is used to classify the data points. So that's what our model is learning as a result of knowing where that boundary is.
- It can take an example that is given to it and decide in each case which class it falls into.

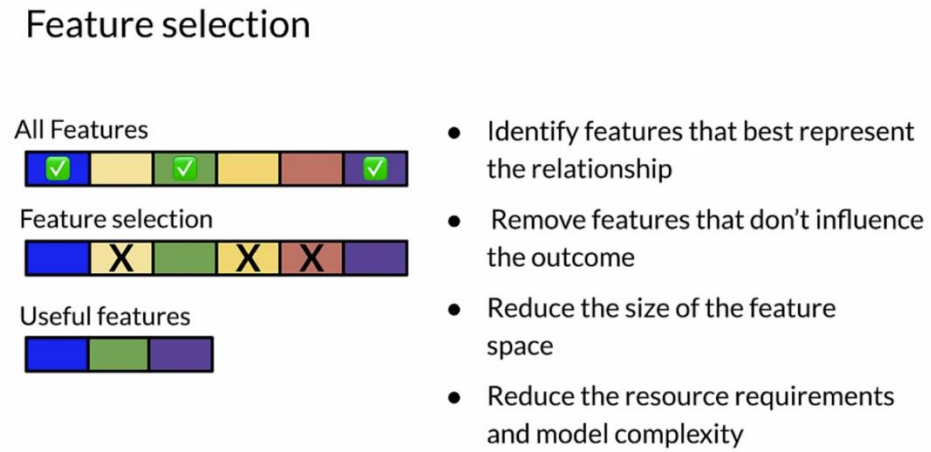


- So feature space coverage is important.
- The examples that you give to your data to your model to train and for evaluation have to be representative of the examples that you're going to see when you serve your model.
- In other words, that region of your feature space has to be covered when you train and evaluate your model
- So that means the same numerical ranges for the same classes and you need similar characteristics for image data as well.
- Similarly for vocabulary. In this case we have syntax and semantics that we have to consider as well for natural language problems.

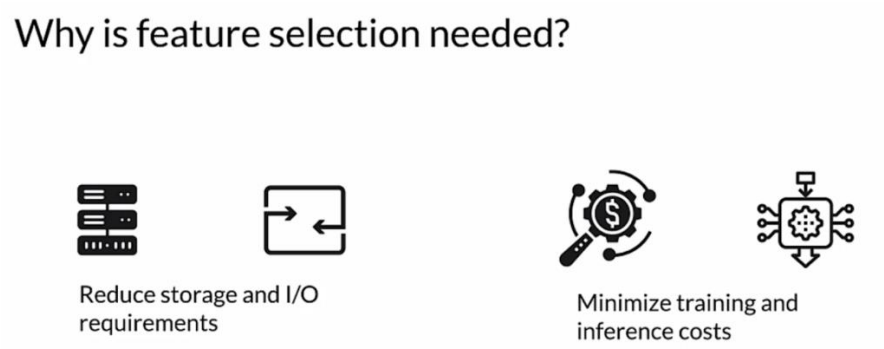


- So we need to ensure that we covered the same space. If we're doing time series problem, we need to consider seasonality trend and drift. And that includes the serving data
- We may have new values that we find in new labels, and that's going to influence the results and cause concept drift. We need to be aware of that and design processes to deal with that.
- That also means we need **continuous monitoring**, which is going to be a key for success in these situations. Remember as we've talked about before, the world changes and our model only learns one version of the world. So when the world has changed, we need to update our model to the new world.

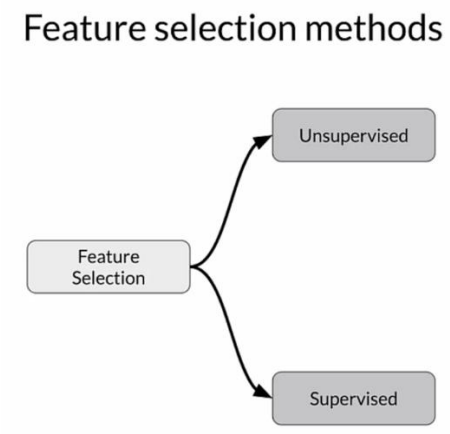
Feature Selection



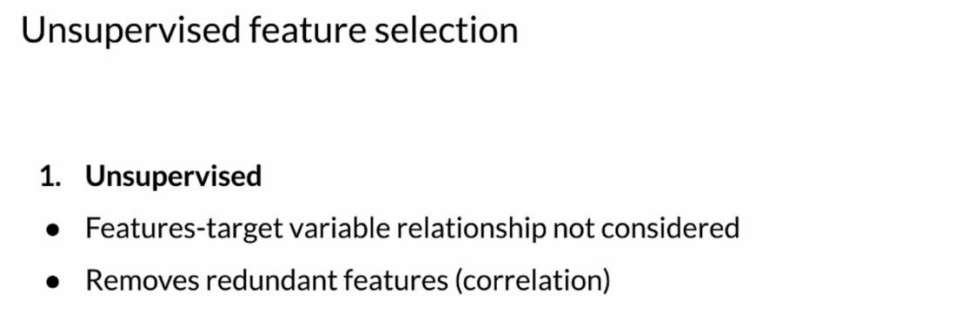
- So we try to select which features we actually need, and eliminate the features that we don't need.
- That gives us a smaller set of features, which are useful features i.e. the ones that we have selected.
- Feature selection identifies the features that best represent the relationship between the features and the target that we're trying to predict. It removes features that don't influence the outcome.
- That reduces the size of the feature space. So remember, the number of features in the feature vector defines the size of a space. Every time we add a feature, the space increases exponentially. So we want to try to reduce the number of features.
- That reduces the resource requirements for processing our data, and the model complexity as well.



- We want to reduce storage and I/O requirements. Less features means smaller data.
- And we want to minimize the training and inference costs, because we may be serving millions of requests once we have trained our model, and each one of those inference costs a certain amount to serve.



- First of all, you can do unsupervised feature selection, meaning you don't have to or don’t need to account for the labels, or supervised feature selection, where it is going to use the information in the labels.



- For unsupervised feature selection, it takes the features, and the target variable relationship is not considered.
- It removes the redundant features, which in this unsupervised method, really means looking for **correlation**.
- When you have two features or more than two features that are highly correlated, you really only need one of those, and you're going to select just the one that will probably give you the best result.

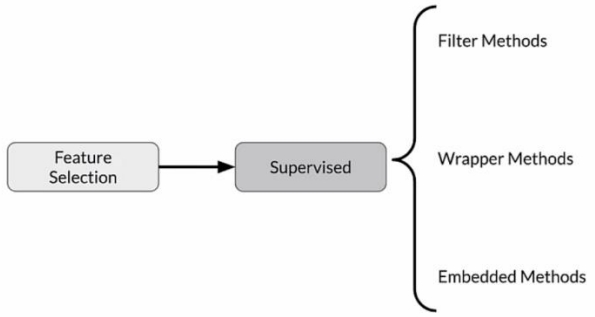
Supervised feature selection

2. Supervised

- Uses features-target variable relationship
- Selects those contributing the most

- For supervised feature selection, it is going to use the target relationship, meaning it is going to lookat the the relationship between each of the features, and the target label.
- It's going to select those features, that contribute most to correctly predicting the target.

Supervised methods

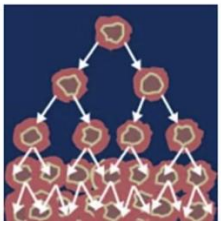


- For supervised feature selection, there's different methods to do that: filter methods, wrapper methods, and embedded methods.

Practical example

Feature selection techniques on Breast Cancer Dataset (Diagnostic)

Predicting whether tumour is benign or malignant.



- For a practical example, we're going to work with a dataset from breast cancer diagnostics. This is going to be the data set we're going to use for looking at several different kinds of feature selection
- We're going to try to predict whether the tumor is benign or malignant

Feature list

id	diagnosis	radius-mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	
842302	M	17.99	10.38	122.8	1001.0	0.1184	0.2776	
concavity_mean	concavepoints_mean	symmetry_mean	fractal_dimension_mean	radius_se	texture_se	perimeter_se	area_se	Irrelevant features
0.3001	0.1471	0.2419	0.07871	1.095	0.9053	8.589	153.4	
smoothness_se	compactness_se	concavity_se	concavepoints_se	symmetry_se	fractal_dimension_se	radius-worst	texture_worst	
0.0064	0.049	0.054	0.016	0.03	0.006	25.38	17.33	
perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concavepoints_worst	symmetry_worst	fractal_dimension_worst	Unnamed:32
184.6	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.1189	NaN

- Here is our feature list, and as you can see we've created and assigned an id for our example.
- We see an irrelevant feature called Unnamed:32.
- If a feature has many missing values, that's often a clue that you have an irrelevant feature.

Performance evaluation

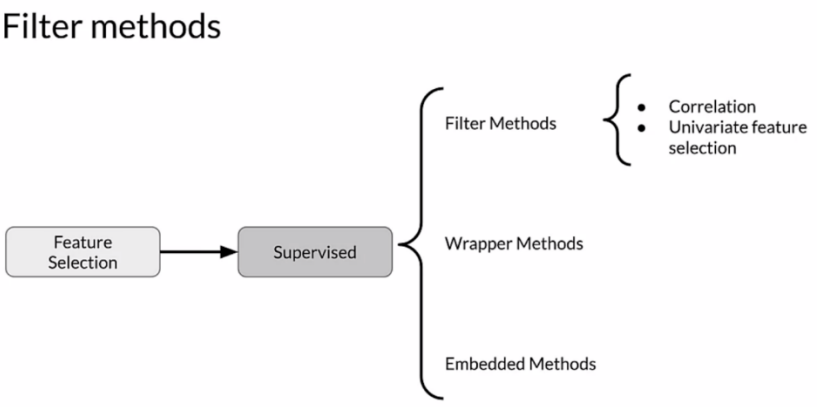
We train a **RandomForestClassifier** model in sklearn.ensemble on selected features

Metrics (`sklearn.metrics`):

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488

- And for performance evaluation, will start as a baseline value. We're going to use a random forest classifier, using sklearn to work with our selected features.
- So, this is our baseline metrics values when we use all 30 features.

Filter Methods



- For filter methods, we're primarily using correlation to look for the features that contain the information that we're going to use to predict our target. Univariate feature selection is also very often used for efficiency

Filter methods

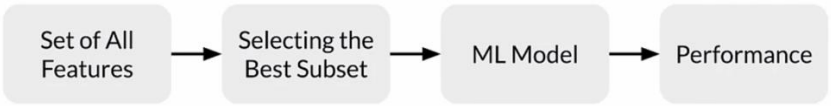
- Correlated features are usually redundant
  - Remove them!

Popular filter methods:

- Pearson Correlation
  - Between features, and between the features and the label
- Univariate Feature Selection

- Let's take a look at some filter methods. Correlated features are usually redundant as we just talked about.
- When you have to features that are highly correlated with each other, that usually is an indicator that they're redundant. You want to choose one of those so you remove the other one.
- Popular filter methods include Pearson correlation. These are different ways to do correlation. Pearson correlation is one. That's correlation between features, and between the features and the label. That's why it's a supervised method.
- You can also do univariate feature selection, which is very common

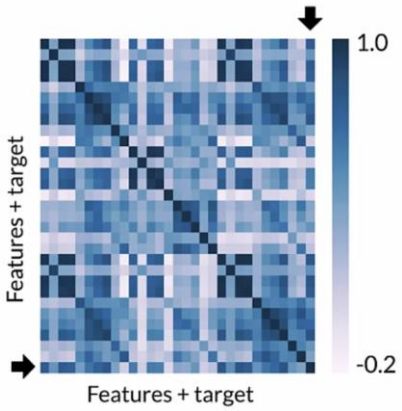
Filter methods



- We are going to start with all of the features, and we're going to select the best subset
- We're going to give those features to our model and that's going to give us our performance for the model with this subset of our features.

Correlation matrix

- Shows how features are related:
  - To each other (Bad)
  - And with target variable (Good)
- Falls in the range [-1, 1]
  - 1 High positive correlation
  - -1 High negative correlation



- One of the ways to visualize this is really correlation matrix. We can look for features where we can see that there is a correlation between two or more of our features.
- It helps show how features are related, both to each other and with the target variable
- You only want one features that are not correlated with one another, but correlated with the target.
- That's going to fall in a range of correlation between -1 and +1, with +1 representing high positive correlation and -1 representing high negative correlation

Feature comparison statistical tests

- Pearson's correlation: Linear relationships
- Kendall Tau Rank Correlation Coefficient: Monotonic relationships & small sample size
- Spearman's Rank Correlation Coefficient: Monotonic relationships

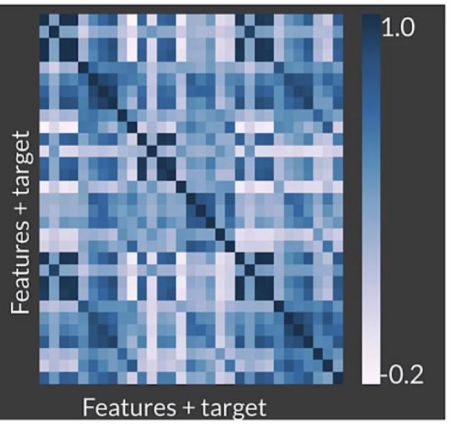
Other methods:

- Mutual information
- F-Test

- Comparing the different tests, you have Pearson's correlation for linear relationships. That's by far I think the most common.
- Kendall Tau is a rank correlation coefficient, looks at monotonic relationships, and it's usually used with a fairly small sample size for efficiency.
- Spearman's is another one. It's also rank correlation and it also is looking at monotonic relationship. Other methods include mutual information, F-test, and chi-square

Determine correlation

```
# Pearson's correlation by default
cor = df.corr()


plt.figure(figsize=(20,20))
# Seaborn
sns.heatmap(cor, annot=True, cmap=plt.cm.PuBu)
plt.show()
```



- Looking at how we would do this in code, let's just take a look using Pearson's correlation.
- This is going to use pandas. We have a data frame and we're using the Pearson correlation.corr method. That gives us our correlation and we can then draw our heat map with seaborn and that gives us our correlation matrix.

Selecting features

```
cor_target = abs(cor["diagnosis_int"])

# Selecting highly correlated features as potential features to eliminate
relevant_features = cor_target[cor_target>0.2]
```

- We then use the absolute correlation values and select highly correlated features to eliminate

Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488
Correlation	**21**	**0.974206**	**0.973684**	**0.953488**	**0.97619**	**0.964706**

- In this case, we've eliminated 9 features because they were correlated. Now, instead of 30, we have 21 features in our feature vector.
- By removing features, mostly almost all our metrics improved.
- That also goes along with the improvements that we made in the compute resources that are required to process 21 features instead of 30.

Univariate feature selection in SKLearn

SKLearn Univariate feature selection routines:

1. **SelectKBest**
2. SelectPercentile
3. GenericUnivariateSelect

Statistical tests available:

- Regression: `f_regression, mutual_info_regression`
- Classification: `chi2, f_classif, mutual_info_classif`

- Let's look at univariate feature selection. We're going to do that using Sci-kit learn.
- There's SelectKBest, SelectPercentile, and GenericUnivariateSelect, which is fairly generic.
- Some statistical tests we can use with that, there's mutual information and F-tests for regression problems.
- For classification, we have chi-squared and there's a version of the F-test for classification and similarly for mutual information, there's a version of that for classification.

SelectKBest implementation

```
def univariate_selection():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                          test_size = 0.2,stratify=Y, random_state = 123)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

 min_max_scaler = MinMaxScaler()
 Scaled_X = min_max_scaler.fit_transform(X_train_scaled)
 selector = SelectKBest(chi2, k=20) # Use Chi-Squared test
 X_new = selector.fit_transform(Scaled_X, Y_train)
 feature_idx = selector.get_support()
 feature_names = df.drop("diagnosis_int",axis = 1 ).columns[feature_idx]
 return feature_names
```

- Let's look at how that gets implemented in code. We've got a function that we're going to define for univariate selection.
- Our data set has been split into training and test, and then that's going to be scaled with a standard scalar
- We're going to use the MinMaxScalar as well to give us a scaled x
- For our selector, we're going to use **SelectKBest** and we're going to use that with the chi-squared test and that's going to look for the **20 best features** here.

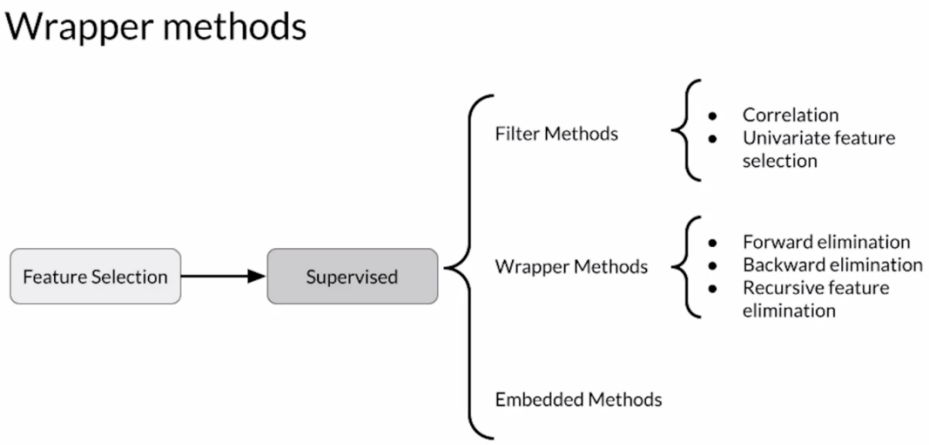
- We're going to fit that transform, we're going to get the features that it selects, and we're going to drop the other features in our original data set.

Performance table

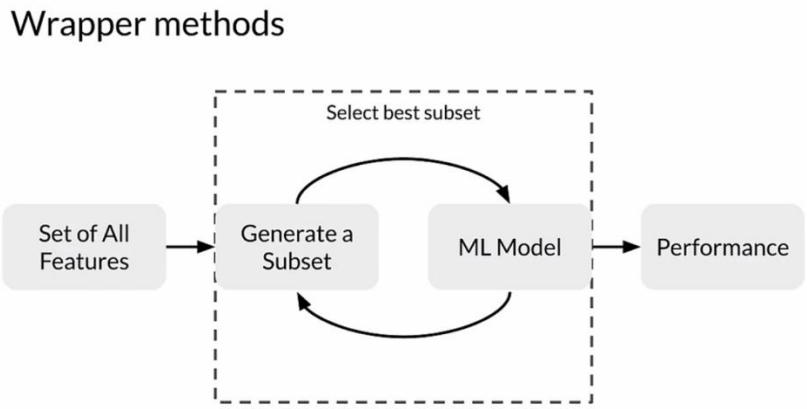
Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488
Correlation	21	0.974206	0.973684	0.953488	0.97619	0.964706
Univariate (Chi^2)	**20**	**0.960317**	**0.95614**	**0.91111**	**0.97619**	**0.94252**

- For the univariate test using chi-squared, we asked for 20 features.
- Most of the metrics dropped by a little bit. This means the correlation method still gave us the best result

Wrapper Methods



- Wrapper methods work all differently. It is still a supervised method, but we're going to use this with a model, and there's different ways to do it. But basically it's a search method against the features that you have **using a model** as a measure of their effectiveness.
- We can do it through forward elimination, backward elimination or recurrent feature elimination.



- We start with all of our features, regenerate a subset of those features, and that gets given to our model.
- The results generated from that model is then used to generate the next subset. That becomes this feedback loop to select the best subset of our features using our model as a measure. That gives us the performance of the final best subset that is selected

Wrapper methods

Popular wrapper methods

1. Forward Selection
2. Backward Selection
3. Recursive Feature Elimination

- There's different wrapper methods: forward selection and backwards selection and recursive feature selection or recursive feature elimination
- These can all be used to select a subset through each iteration of that feedback loop.

Forward selection

1. Iterative, greedy method
2. Starts with 1 feature
3. Evaluate model performance when **adding** each of the additional features, one at a time
4. Add next feature that gives the best performance
5. Repeat until there is no improvement

- We start with one feature (it's based on a greedy algorithm), and then we evaluate the model performance as we add features one at a time.
- We're gradually building up this feature vector, one feature at a time, starting with just one feature.
- We try to add the next feature that gives the best performance, and we're going to measure that to see what the result is.
- We keep repeating this until there's no improvement and then we know that we've generated the best subset of our feature.

Backward elimination

1. Start with all features
2. Evaluate model performance when **removing** each of the included features, one at a time
3. Remove next feature that gives the best performance
4. Repeat until there is no improvement

- Backward elimination starts with all of the features and evaluates the model performance when removing feature. It's exactly what you might think from the name one at a time. We remove the next feature, trying to get to better performance with less features and we keep doing that until there's no improvement.

Recursive feature elimination (RFE)

1. Select a model to use for evaluating feature importance
2. Select the desired number of features
3. Fit the model
4. Rank features by importance
5. Discard least important features
6. Repeat until the desired number of features remains

- For recursive feature elimination, we select a model to use for evaluating feature importance.
- We select the desired number of features and we fit them, and then we rank the features by importance.
- We need to have a method of assigning importance to those features and then we discard the least important features.
- We keep doing that until we get down to the number of features that we're looking to target.
- **An important aspect of this is that we need to have a measurement of feature importance in our model and not all models are able to do that**

Recursive feature elimination

```
def run_rfe():

  X_train, X_test, y_train, y_test = train_test_split(X,Y, test_size = 0.2, random_state = 0)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  model = RandomForestClassifier(criterion='entropy', random_state=47)
  rfe = RFE(model, 20)
  rfe = rfe.fit(X_train_scaled, y_train)
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[rfe.get_support()]
  return feature_names

rfe_feature_names = run_rfe()

rfe_eval_df = evaluate_model_on_features(df[rfe_feature_names], Y)
```

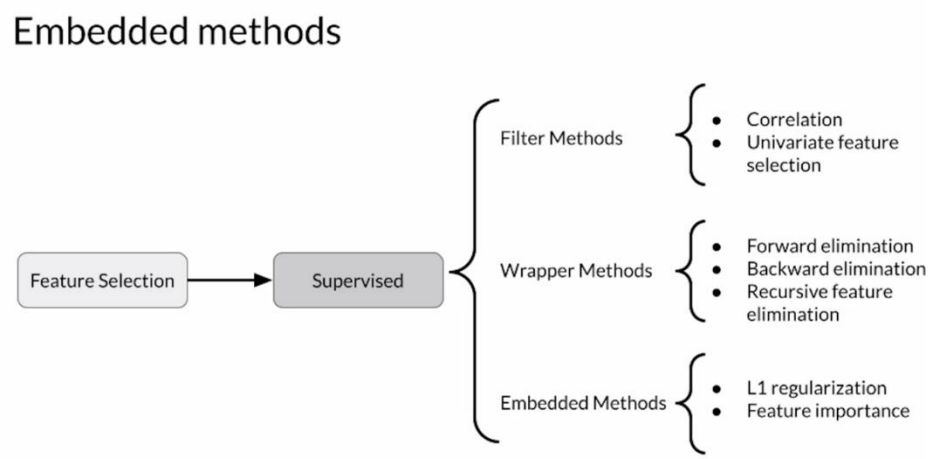
- This is what the code might look like.
- Again, we're pulling in our data and splitting it with train and test, and pulling out the labels from our data
- We have X and Y, why you're going to scale it because it's always a good idea.
- Then we're going to use a random forest classifier. A random forest classifier is one of the model types where we **can get the feature importance**. In this case we're using **entropy** as the metric and we're initializing it with a random state.
- We're going to apply our random feature elimination by trying to get to 20 features with our model that gets fitted. That's going to do that elimination and that results in the list of features that we've selected. Then we're going to use that to do the evaluation to get the final value.

Performance table

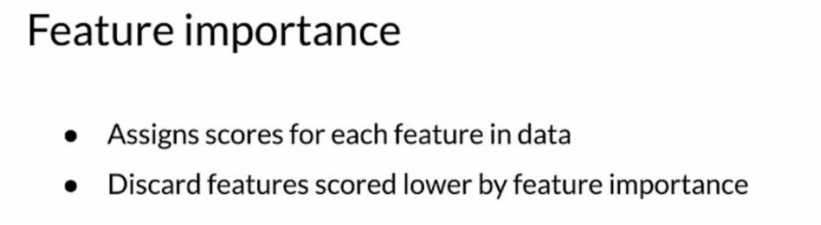
Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.96491	0.931818	0.97619	0.953488
Correlation	21	0.97420	0.97368	0.9534883	0.97619	0.964705
Univariate (Chi^2)	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	**20**	**0.97420**	**0.97368**	**0.953488**	**0.97619**	**0.964706**

- The accuracy was quite a bit better than it was for univariate, in fact, it's as good as correlation.
- The AUC also as good as correlation, which is great.
- For recall, exactly the same, but the F1 score is actually slightly better.
- We've gotten to fewer features, we did it with 20 features, instead of 21 for correlation. That recursive feature elimination is now our best result.

Embedded Methods



- Let's look at Embedded Methods.
- **Embedded methods combine the best of both worlds (filter and wrapper methods)**, because they are faster than wrapper methods (since wrapper methods are based on the greedy algorithm and solutions are slow to compute), and they are more efficient than filter methods (since filter methods suffer from inefficiencies as they need to look at all the possible feature subsets)
- So L1 or L2 regularization is essentially an embedded method for doing feature selection. Feature importance is another method.
- Both of these are highly connected to the model that you're using. So both these regularization and feature importance are really sort of an intrinsic characteristic of the model that you're working with.



- For regularization, we're talking about weighting each feature in the data.

- It discards features often by setting those weights to zero or near zero. And that's going to eliminate the features that we're talking about, essentially based on the feature importance.

Feature importance with SKLearn

- Feature Importance class is in-built in Tree Based Models (eg., `RandomForestClassifier`)
- Feature importance is available as a property `feature_importances_`
- *We can then use* `SelectFromModel` *to select features from the trained model based on assigned feature importances.*

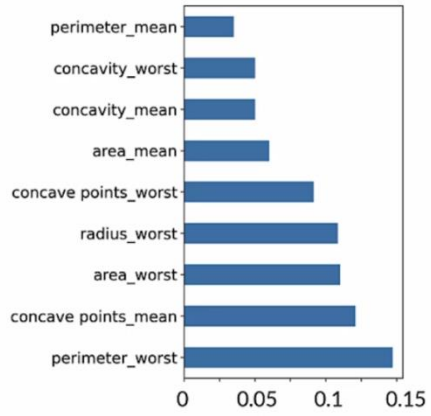
- So looking at that in SKLearn, if we look at Feature Importance class, that's built into the Tree Based Model.
- Again, we're still using the RandomForestClassifier that we've been using all along. That's one of the model types that does include feature importance. It's available as a property of that model
- We can use SelectFromModel to select the features from the train model based on the assigned feature importances.

Extracting feature importance

```
def feature_importances_from_tree_based_model_():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2,
                                                      stratify=Y, random_state = 123)
  model = RandomForestClassifier()
  model = model.fit(X_train,Y_train)

  feat_importances = pd.Series(model.feature_importances_, index=X.columns)
  feat_importances.nlargest(10).plot(kind='barh')
  plt.show()

  return model
```

- How does that look in code? Well, we're going to define a function. Feature importance is from Tree Based Model
- Our data hs been split into training and test, and we've separated our labels.
- We're going to use our RandomForestClassifier as our model and we're going to fit it.
- Then we're going to pull out the feature importances here. That's going to give us a Pandas series, from which we're going to select the 10 best in terms of feature importance

Feature importance plot



- Here's the plot showing the 10 best features based on feature importance calculated in the model

Select features based on importance

```
def select_features_from_model(model):


  model = SelectFromModel(model, prefit=True, threshold=0.012)

  feature_idx = model.get_support()
  feature_names = df.drop("diagnosis_int",1 ).columns[feature_idx]
  return feature_names
```

- We're going to go back and select from our model. That's going to give us our model and we're going to use get_support to get the indexes for those features.
- Then we're going to drop the other features from our feature vector, and that returns us the names of the features that we're keeping.

Tying together and evaluation

```
# Calculate and plot feature importances
model = feature_importances_from_tree_based_model_()

# Select features based on feature importances
feature_imp_feature_names = select_features_from_model(model)
```

- So tying those together, we've got feature importance from tree based model that that's going to give us the feature importances and plot them.
- Then we're going to select the features that we're going to keep, and that's going to give us our performance.

Performance table

Method	Feature Count	Accuracy	ROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.964912	0.931818	0.9761900	0.953488
Correlation	21	0.97420	0.973684	0.953488	0.9761904	0.964705
Univariate Feature Selection	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	20	0.9742	0.973684	0.953488	0.97619	0.964706
Feature Importance	14	0.96726	0.96491	0.931818	0.97619	0.953488

- So in this case we've selected 14 features and looking at the metrics for them, they're not quite as good
- Recursive Feature Elimination is really still our best result, although Feature Importance was able to get us down to a smaller number of features

Review

- Intro to Preprocessing
- Feature Engineering
- Preprocessing Data at Scale
  - TensorFlow Transform
- Feature Spaces
- Feature Selection
  - Filter Methods
  - Wrapper Methods
  - Embedded Methods

References

- Mapping raw data into feature
- Feature engineering techniques
- Scaling
- Facets
- Embedding projector
- Encoding features
- https://www.tensorflow.org/tfx/guide#tfx_pipelines
- https://ai.googleblog.com/2017/02/preprocessing-for-machine-learning-with.html
- Breast Cancer Dataset