Machine Learning Data Lifecycle in Production

MLOps Specialization Course 2

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

Implement feature engineering, transformation, and selection with TensorFlow Extended by encoding structured and unstructured data types and addressing class imbalances

- Define a set of feature engineering techniques, such as scaling and binning
- Use TensorFlow Transform for a simple preprocessing and data transformation task
- Describe feature space coverage and implement different feature selection methods
- Perform feature selection using scikit-learn routines and ensure feature space coverage

Compiled By	**Peter Boshra**
LinkedIn	**https://www.linkedin.com/in/peterboshra/**
GitHub	**https://github.com/PeterBushra**
Email	**Dev.PeterBoshra@gmail.com**

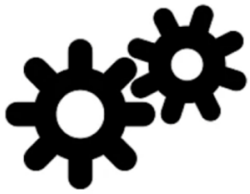
2.2 Week 2: Feature Engineering, Transformation and Selection

2.2.1 Feature Engineering

"coming up with features is difficult, time consuming and requires expert knowledge. Applied machine learning often requires careful engineering of the features and data set." – Andrew Ng

Outline

- Squeezing the most out of data
- The art of feature engineering
- Feature engineering process
- How feature engineering is done in a typical ML pipeline

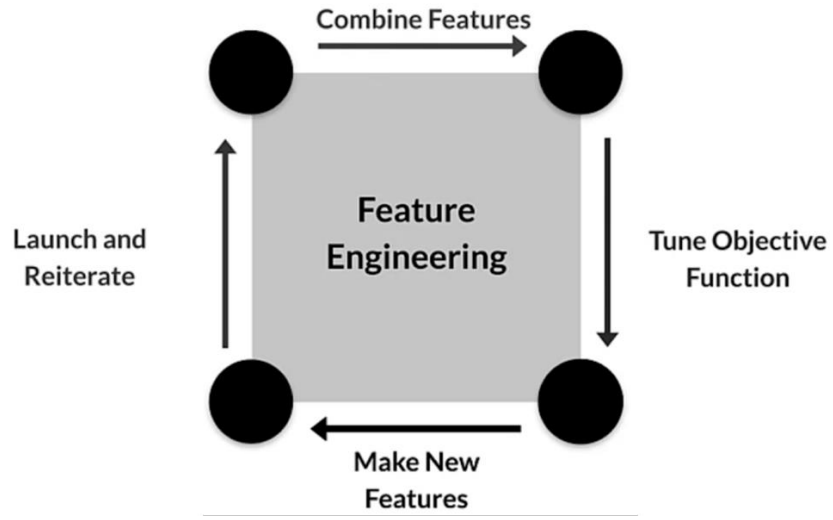


Squeezing the most out of data

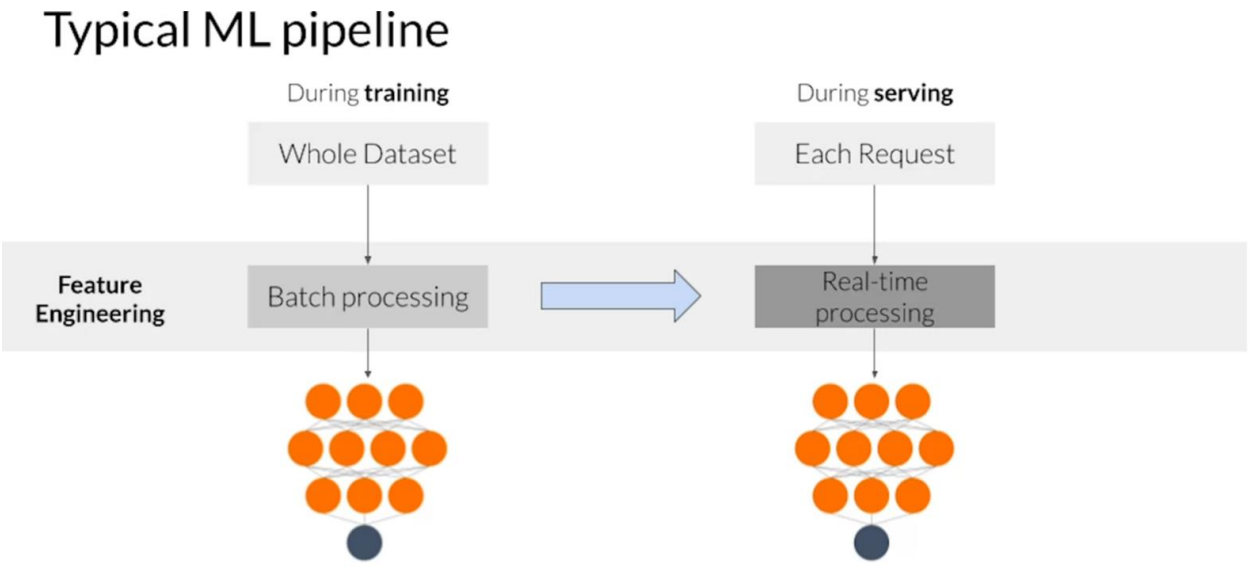
- Making data useful before training a model
- Representing data in forms that help models learn
- Increasing predictive quality
- Reducing dimensionality with feature engineering

- squeeze information out of our data. So, Emma models usually require some data preprocessing to improve training and you should probably by now have been training models that this is very familiar to you.
- 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.
- techniques for selecting and transforming the input data are key to increase the predictive quality of the models and 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.
- 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, if possible, 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. So 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 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,
- his tends to be an iterative process that gradually improves.



- 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. However, 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.
- 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.

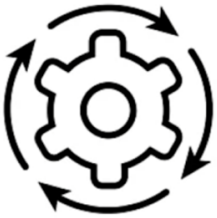
Key points

- Feature engineering can be difficult and time consuming, but also very important to success
- Squeezing the most out of data through feature engineering enables models to learn better
- Concentrating predictive information in fewer features enables more efficient use of compute resources
- Feature engineering during training must also be applied correctly during serving

2.2.1.1 Preprocessing Operations

Outline

- Main preprocessing operations
- Mapping raw data into features
- Mapping numeric values
- Mapping categorical values
- Empirical knowledge of data



Main preprocessing operations

Data cleansing

Feature tuning

Representation transformation

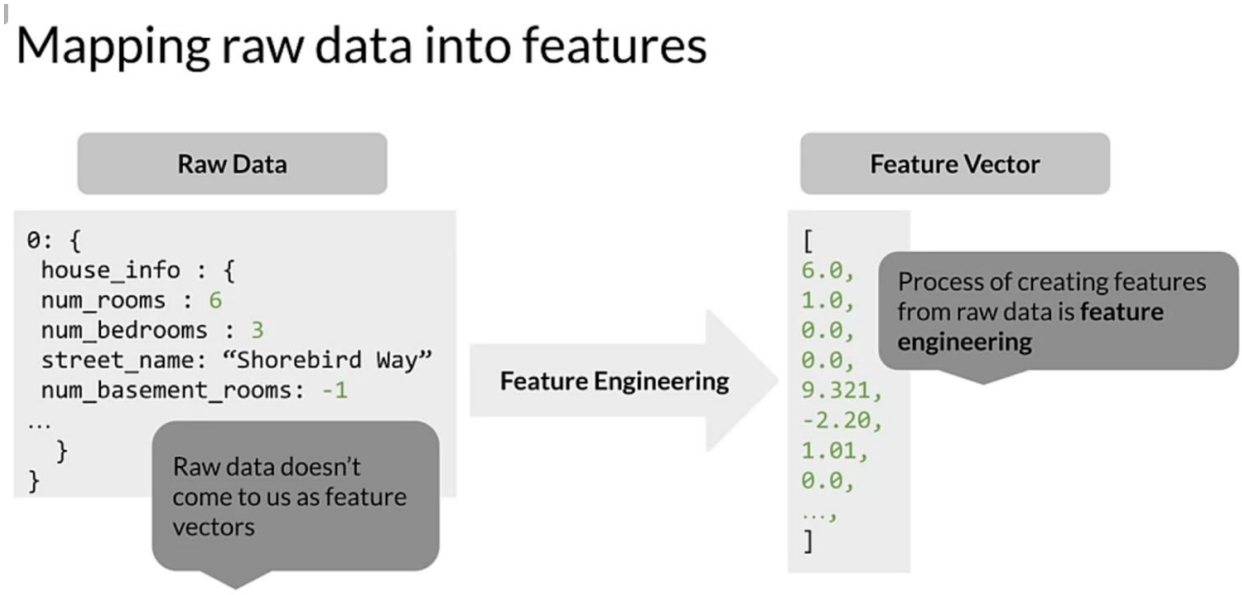
Feature extraction

Feature construction

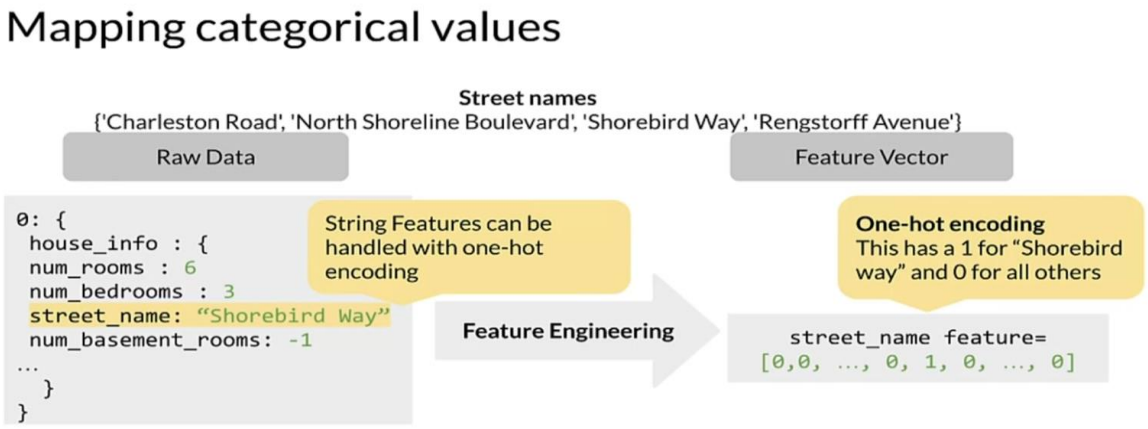
this is not going to be an exhaustive list There's a lot of things you can do to data.

- Data cleansing, which in broad terms consists in eliminating or correcting erroneous data.
- You'll often need to perform transformations on your data, so scaling or normalizing your numeric values, for example. 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 represents.
- Feature construction can be used to create new features by using several different techniques.



We're looking at a house. This is data from a house, but this is only what we start with. The raw data. Feature Engineering because it's in performing an analysis of the raw data and then creating a feature vector from it. For example, integer data can be mapped to floats, and numerical data can be normalized. One-hot vectors can be created from categorical values. Feature Engineering creates features from raw data and you're probably familiar with this.



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)
```

Empirical knowledge of data

Text - stemming, lemmatization, TF-IDF, n-grams, embedding lookup

Images - clipping, resizing, cropping, blur, Canny filters, Sobel filters, photometric distortions

But you 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 and that really focuses on the semantic value of the words. That's something we know as data scientists or Machine Learning Engineers about working with data. Images are similar. There's things that we will know about how we can improve the predictive qualities of images.

Key points

- Data preprocessing: transforms raw data into a clean and training-ready dataset
- Feature engineering maps:
  - Raw data into feature vectors
  - Integer values to floating-point values
  - Normalizes numerical values
  - Strings and categorical values to vectors of numeric values
  - Data from one space into a different space

2.2.1.2 Feature Engineering Techniques

Outline

- Feature Scaling
- Normalization and Standardization
- Bucketizing / Binning
- Other techniques

Feature Engineering takes a variety of forms, normalizing, and scaling features, and coding categorical values and so forth. We have scaling and normalizing and standardizing. We can also do groupings, so that could be bucketizing. Where things like for text, we could do a bag of words. 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.

Feature engineering techniques

Numerical Range
- Scaling
- Normalizing
- Standardizing

Grouping
- Bucketizing
- Bag of words

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

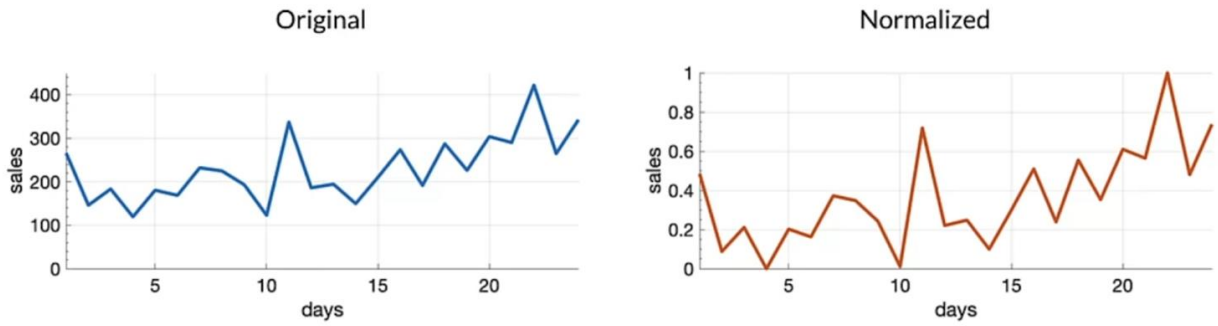
Scaling converts to standard range and different techniques to it differently. For example, a gray-scale image pixel intensity is typically expressed in the raw data as a number between 0-255, and then that's usually re-scale to negative one to one.

Normalization

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

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

Normalization



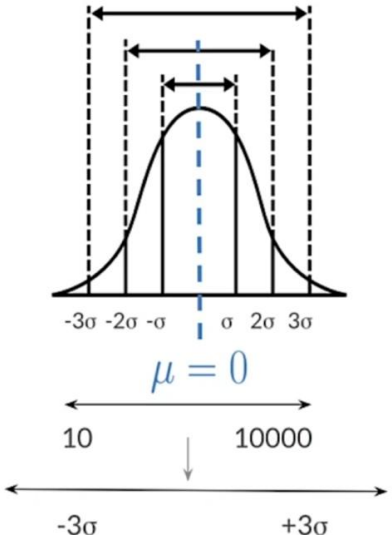
Here's the standard formula for normalizing something. It's also called min-max. You're taking your X, you're subtracting the minimum value of that feature. You have to make a full pass over your data set to get that minimum value, and then you subtract that and then you divide by the maximum value minus the minimum value. You're going to need to make a full pass to get those numbers. 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**. Doesn't always have to be true, but typically that's a good assumption to start with. A good rule of thumb. If you're working with data that you know is not Gaussian, or a normal distribution, then normalization is a good technique to start with.

Standardization (z-score)

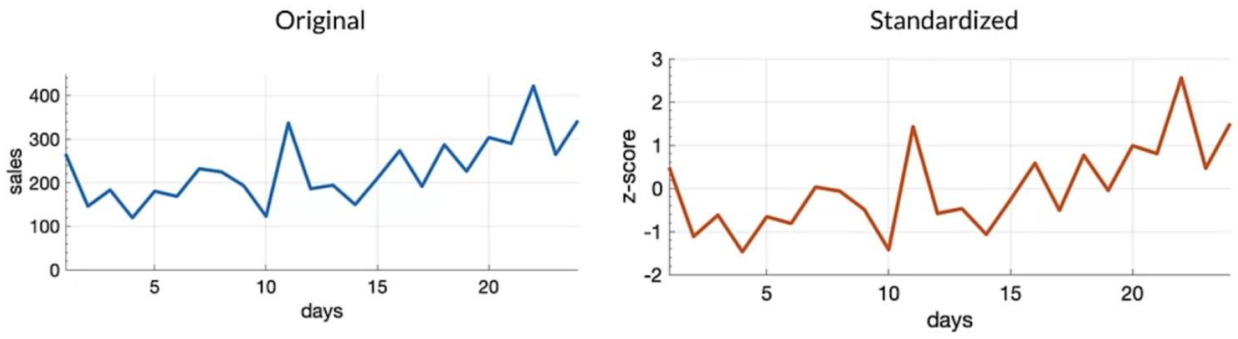
- Z-score relates the number of standard deviations away from the mean
- Example:

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

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

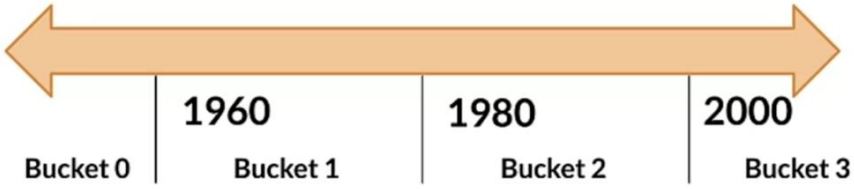


Standardization (z-score)



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. The example here is using, , and you're 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. If the original looked like this, again, a standardized value of that might look like this. 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. It's a little bit less bounded transformation than a normalization is. **But there are some advantages to it, that your data is a normal distribution**, then a standardization technique is a good place to start, it's a good rule of thumb to start with for your numerical features. But I encourage you to try both standardization and normalization and compare the results.

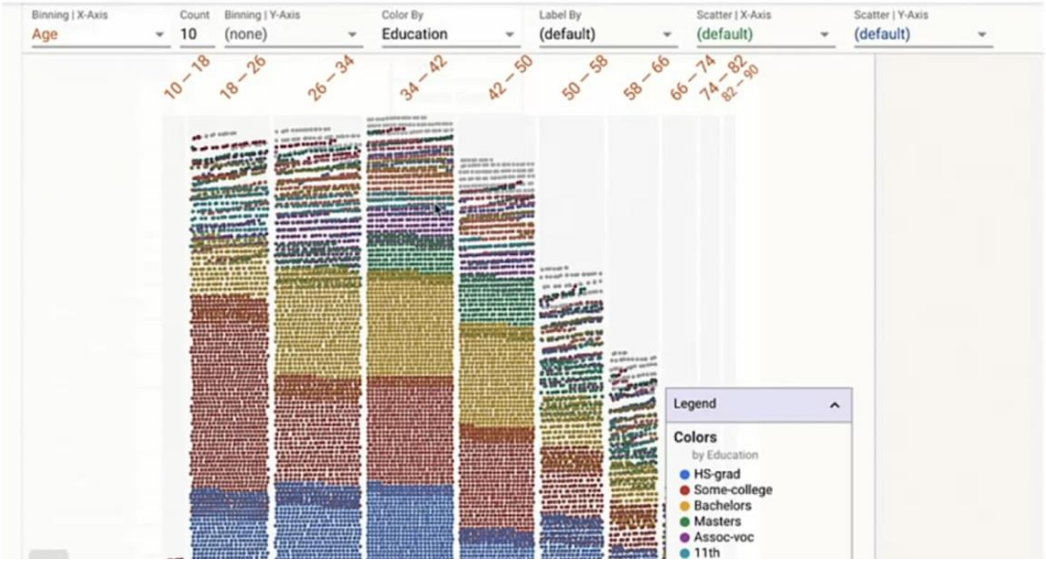
Bucketizing / Binning



Date Range	Represented as...
< 1960	[1, 0, 0, 0]
>= 1960 but < 1980	[0, 1, 0, 0]
>= 1980 but < 2000	[0, 0, 1, 0]
>= 2000	[0, 0, 0, 1]

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 it into different buckets.

Binning with Facets



Other techniques

Dimensionality reduction in embeddings

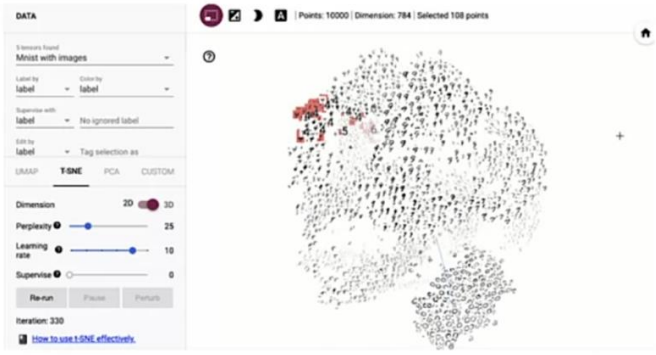
- Principal component analysis (PCA)
- t-Distributed stochastic neighbor embedding (t-SNE)
- Uniform manifold approximation and projection (UMAP)

Feature crossing

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 interesting and has some interesting aspects.

TensorFlow embedding projector

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



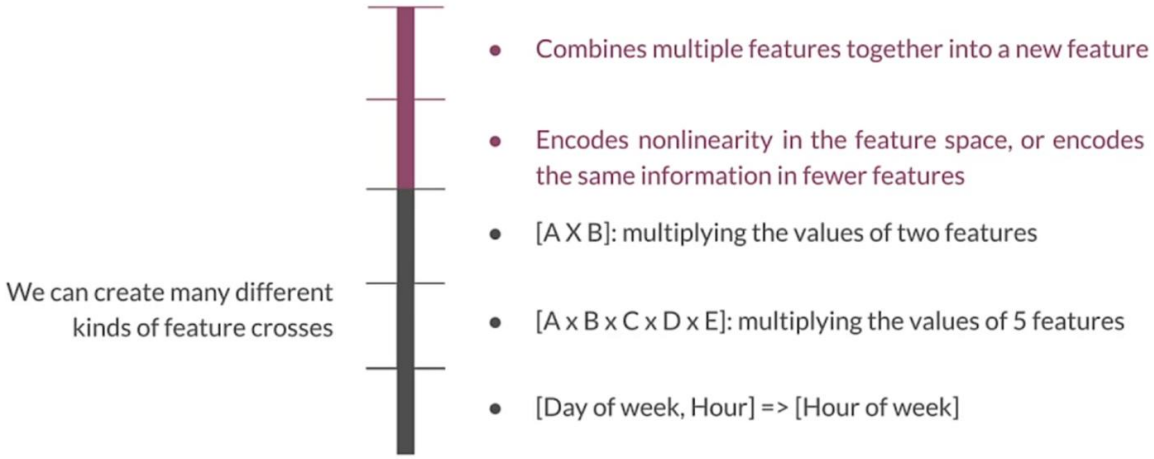
Key points

- Feature engineering:
  - Prepares, tunes, transforms, extracts and constructs features.
- Feature engineering is key for model refinement
- Feature engineering helps with ML analysis

Outline

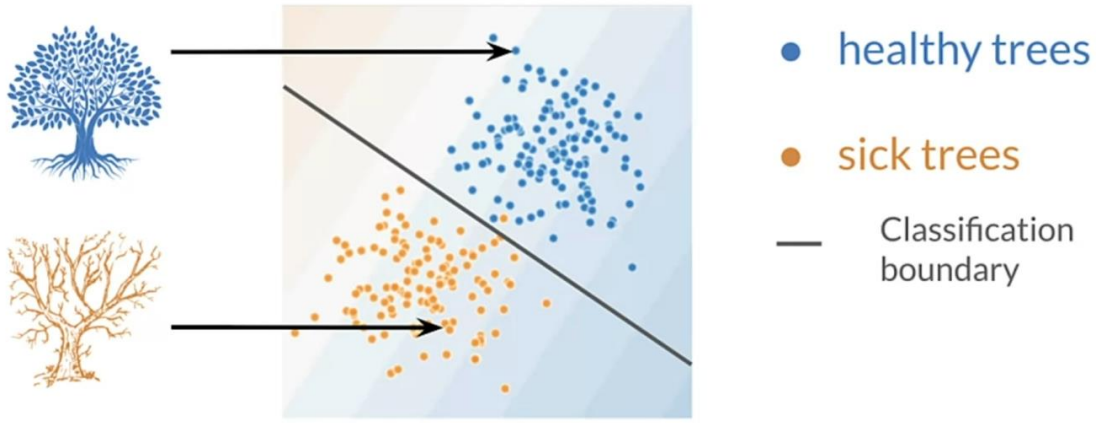
- Feature crosses
- Encoding features

Feature crosses

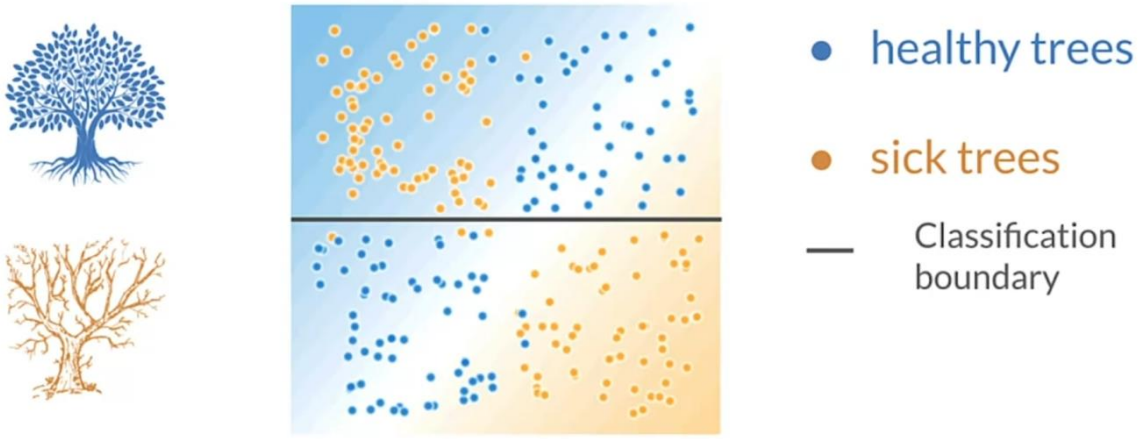


Feature crosses, combine multiple features together into a new feature. That's fundamentally what a feature across. It encodes non-linearity in the feature space, or encodes the same information and fewer features. We can create many different kinds of feature crosses and it really depends on our data. It requires a little bit of imagination to look for ways to try to do that, to combine the features that we have.

Encoding features

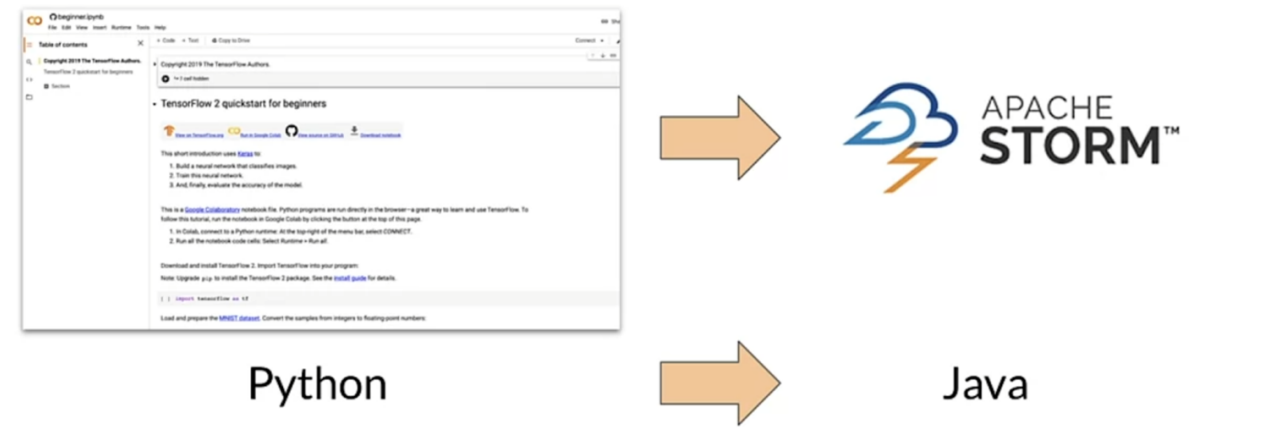


Need for encoding non-linearity

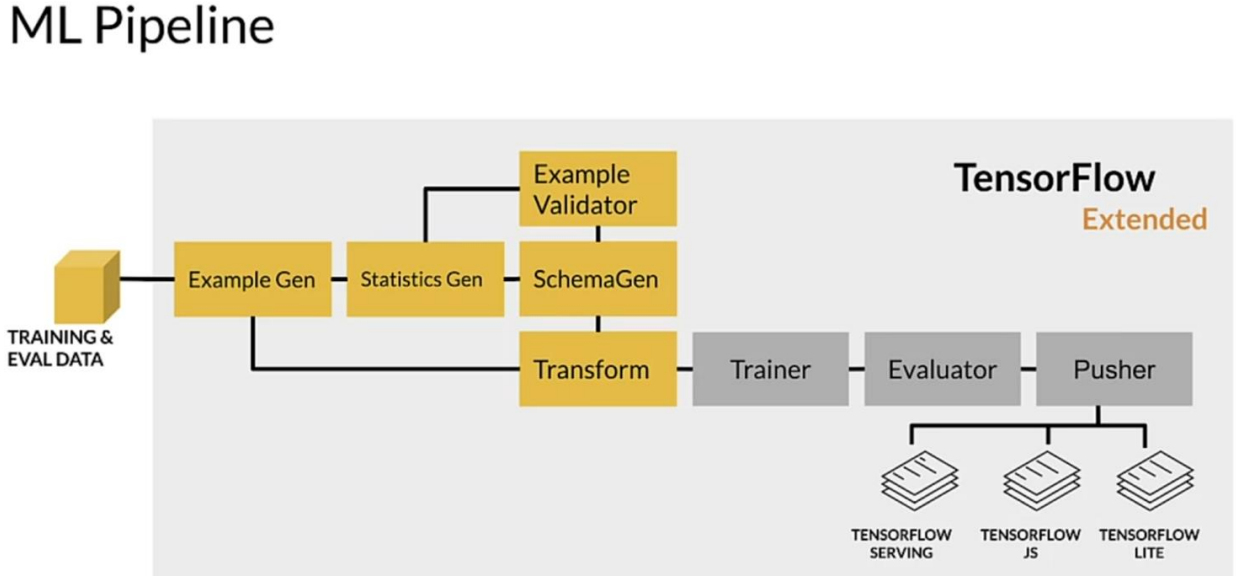


Key points

- Feature crossing: synthetic feature encoding nonlinearity in feature space.
- Feature coding: transforming categorical to a continuous variable.



We were developing our notebooks in python and then when we deployed into storm we had to translate all of the feature engineering that we did into java. Well, as you can imagine, that was not ideal and there were little weird problems that cropped up often very difficult to find doing that transformation. This is not an ideal way to do production machine learning. So what is much better technique is to use a pipeline, a unified framework where you can both train and deploy with consistent and reproducible results.



Outline

- Inconsistencies in feature engineering
- Preprocessing granularity
- Pre-processing training dataset
- Optimizing instance-level transformations
- Summarizing the challenges

Preprocessing data at scale

Real-world models: terabytes of data

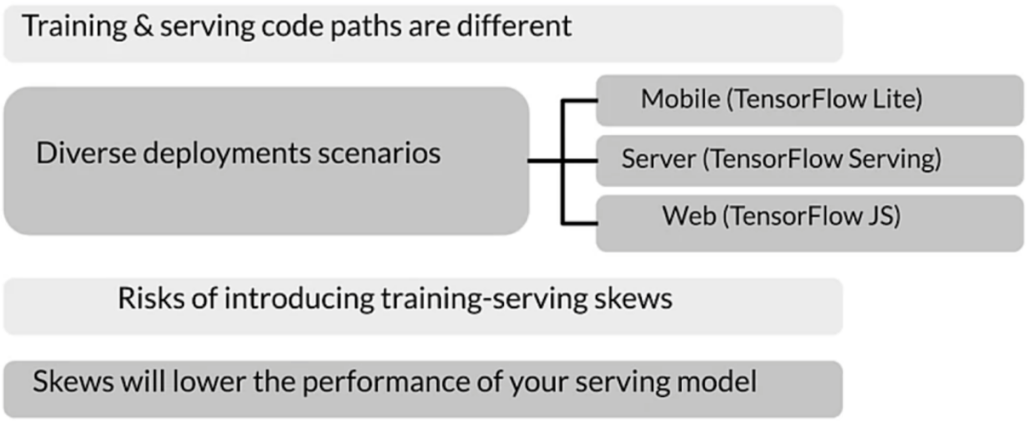
Large-scale data processing frameworks

Consistent transforms between training & serving

preprocessed data at scale, we start with real world models and these could be terabytes of data. So when you're doing this kind of kind of work, you want to start with a subset of your data work out. As many issues as you can think about how that's going to scale up to the terabytes that you you need to actually work with your whole dataset. 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 and the earlier you can start working with those frameworks. The more you work out the issues early with smaller datasets and 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, you are going to have problems and often those problems are very hard to find So inconsistencies and feature engineering the training and serving code paths. When you are training your model, you have code that you're using for training. If you're using different code, like we were we were using python for training and java for serving, that's a potential source of problems. So, and you could have different deployment scenarios. You you might be deploying your model for in different environments. You you could be using the same model, say, in a servant cluster and 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. So mobile, for example, very tight compute resources, servers, you have more luxury, but again, cost is a big factor and in a web browser, that could be deployed on different clients. So you need to think about that as well. You don't want to be too heavy. Though, there's risks in introducing training, serving skews. So that's what happens because of those different code paths between training and serving. If you don't have exactly the same transformation is happening between the two and the best way to do that is used exactly the same code. Then you have a potential problem and that's going to lower the performance of your model. So your model may, completely just error out or give wacky results or it may be just slightly off in certain circumstances and display sensitivity around certain things. Those things are much harder to detect. So there is a notion of the granularity at which you're doing preprocessing. So you need to think about this.

Inconsistencies in feature engineering



you're going to do transformations, both the instance level and a full pass over your data. And depending on the transformation that you're doing, you may be doing it in one or the other. You can usually always do instance level. But full path requires that you have the whole dataset. So for clipping, even for clipping, you need to set clipping boundaries. If you're not doing that through some arbitrary setting of those boundaries, if you're using the data set itself to determine how to clip, then you're going to need to do a full pass. So min max for clipping is is going to be important. Doing a multiplication at the instance level is fine, but scaling well now I'm going to need probably the standard deviation and maybe the min and max. Bucketizing similarly, 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. Expanding features I can probably do that at the instance level. So these two things are different at training time. I have the whole dataset so I can do a full pass, although it can be expensive to do that. At serving time, I get individual examples, so I can only really do instance level. So 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.

When do you transform?

Pre-processing training dataset

Pros	Cons
Run-once	Transformations reproduced at serving
Compute on entire dataset	Slower iterations

How about ‘within’ a model?

Transforming within the model

Pros	Cons
Easy iterations	Expensive transforms
Transformation guarantees	Long model latency
	Transformations per batch: skew

you're going to have to reproduce all those transformations that serving or save off the information that you learned about the data, like the standard deviation. So that you can use that later while you're serving. And there's, slower iterations around this. Each each time you make a change, you've got to make a full pass over the dataset. So you can do this within the model. Transforming within the model has some nice features to there are cons as well. First of all it makes iteration somewhat easier because it's embedded as part of your model and there's guarantees around the transformation that you're doing. But the cons are it can be expensive to do those transforms, especially when, your compute resources are limited. And think about when you do a transform within the model, you're going to do that same transform at serving time. So you may have say GPUs or TPUs when you're training, you may not when you're serving. So there's long model latency, that's when you're serving your model, if you're doing a lot of transformation with it that can slow down the response time for your model and increase latency. And, you can have transformations per batch that that skew that we talked about. If you haven't saved those constants that you need, that could be an issue. 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, which when you're working with terabytes of data. That can be a significant advantage in terms of processing. And there's ways to normalize per batch as well. So you can compute an average and use that and normalization per batch and then do it again for the next batch there will be differences batch to batch. Sometimes that's good in cases. So for

example where you have changes over time in a time series, that can be a good thing but you need to consider that. So normalizing by the average per batch, precompeting the average and using it during normalization, you can use it for multiple batches for example. So this is different ways to try to think about how to work with your data when you, especially when you have a large dataset. You need to think about optimizing the instance level transformations as well because this is going to affect both the training efficiency and you're serving efficiency. So you're going to have accelerators that you need to consider. They may be sitting idle while your CPU is doing transforms. That's something that you want to try to avoid because accelerators tend to be expensive

Why transform per batch?

- For example, normalizing features by their average
- Access to a single batch of data, not the full dataset
- Ways to normalize per batch
  - Normalize by average within a batch
  - Precompute average and reuse it during normalization

Optimizing instance-level transformations

- Indirectly affect training efficiency
- Typically accelerators sit idle while the CPUs transform
- Solution:
    - Prefetching transforms for better accelerator efficiency

So as a solution you can prefetch your your transformations and use your accelerators more efficiently. So by prefetching you can prefetch with your CPU feed your your accelerator your GPU or CPU and try to paralyze that processing. So again this all gets down to cost and how much it costs to train and and the time required as well to train your model 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.

Summarizing the challenges

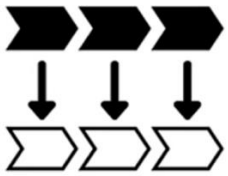
- Balancing predictive performance
- Full-pass transformations on training data
- Optimizing instance-level transformations for better training efficiency (GPUs, TPUs, ...)

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

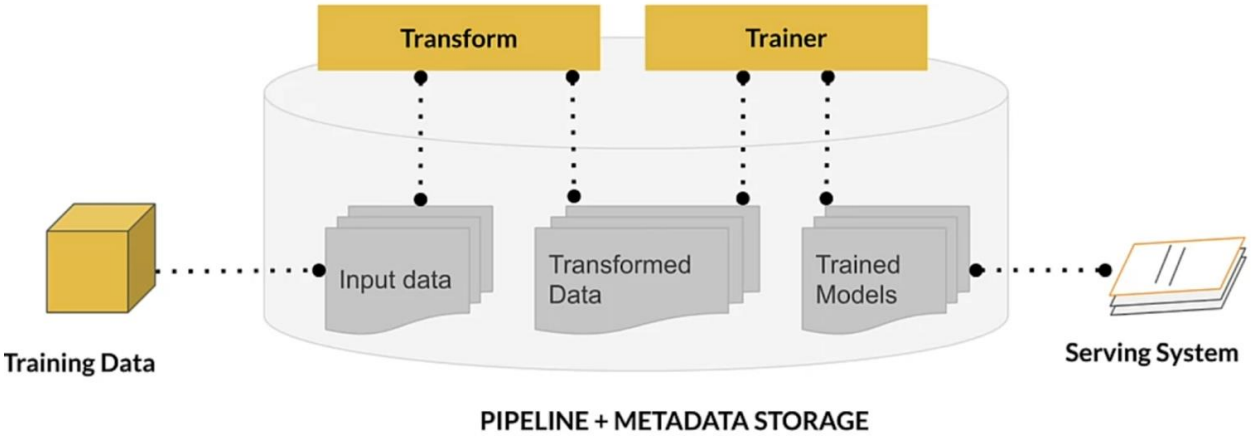
Outline

- Going deeper
- Benefits of using TensorFlow Transform
- Applies feature transformations
- tf.Transform Analyzers



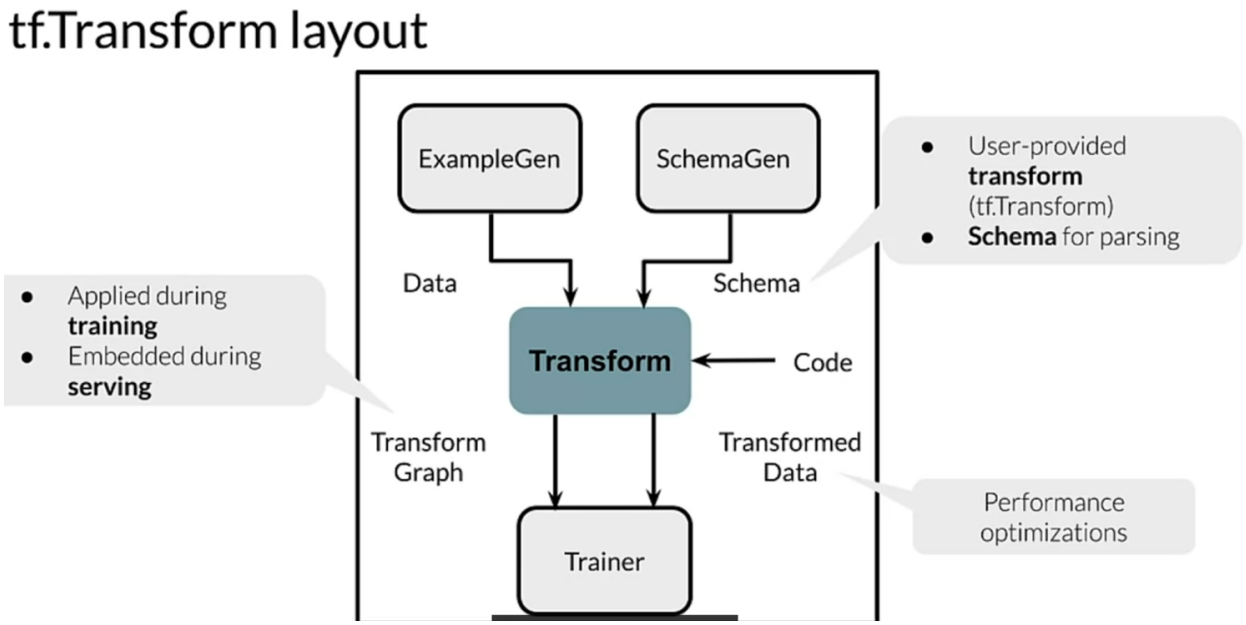
To do pre-processing at scale, we need good tools. TensorFlow Transform is one of the best tools available today

Enter tf.Transform



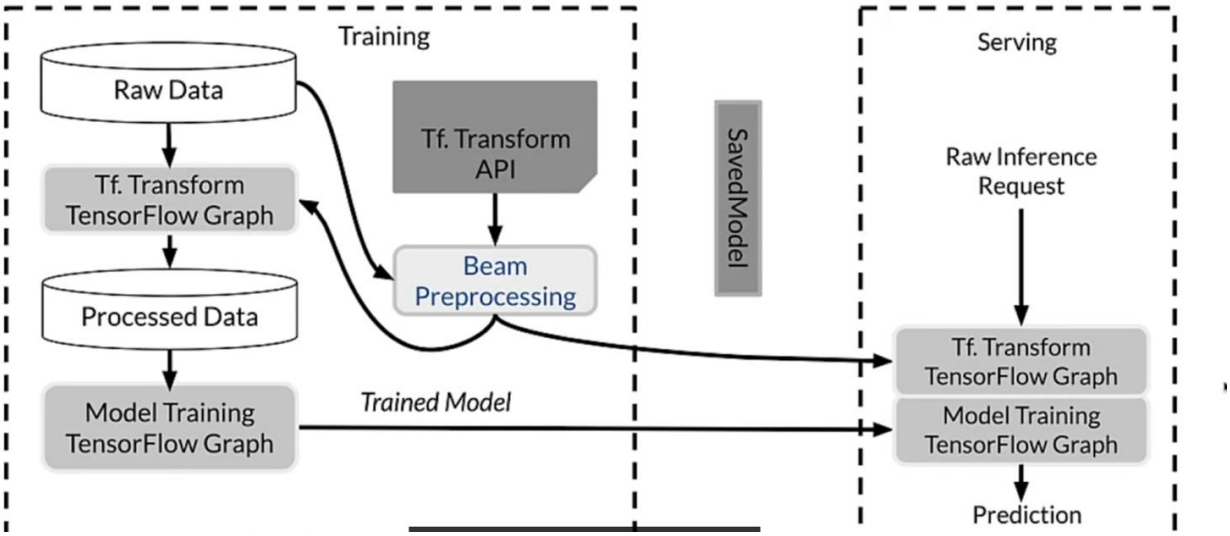
there is METADATA that forms a key role in organizing and managing the artifacts that are produced as data is transformed. One of the reasons that's important, is because, we want to understand the lineage or provenance of those artifacts, and be able to connect them, and find the chain of operations 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. That's 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. These are the Trainer, of course as 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. We've actually split our dataset. We split it with ExampleGen, so ExampleGen does that split. Those get fed to StatisticsGen. These are both TFX components within a TFX pipeline,

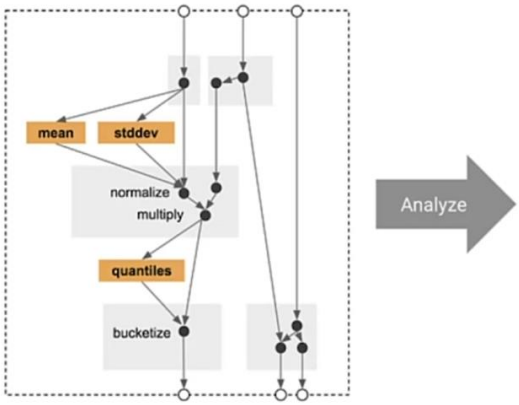


ExampleGen is component, StatisticsGen is a component. StatisticsGen calculates statistics for our data. For each of the features, if they're numeric features, for example, what is the mean of that feature value? What is the standard deviation? The min, the max, so forth. Those statistics get fed to SchemaGen which infers the 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 it looks for problems in our data.

tf. Transform: Going deeper



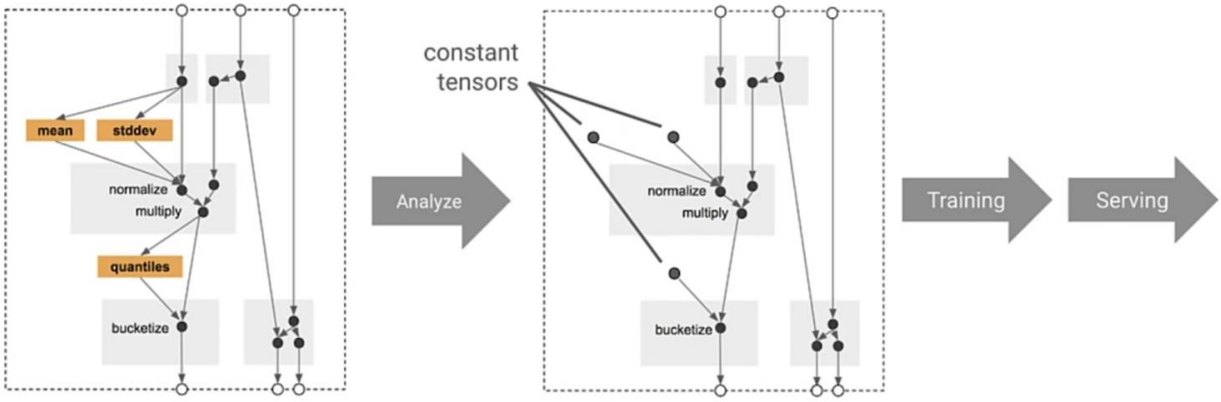
tf.Transform Analyzers



They behave like TensorFlow Ops, but run only once during training

For example:
tft.min *computes the minimum of a tensor over the training dataset*

How Transform applies feature transformations

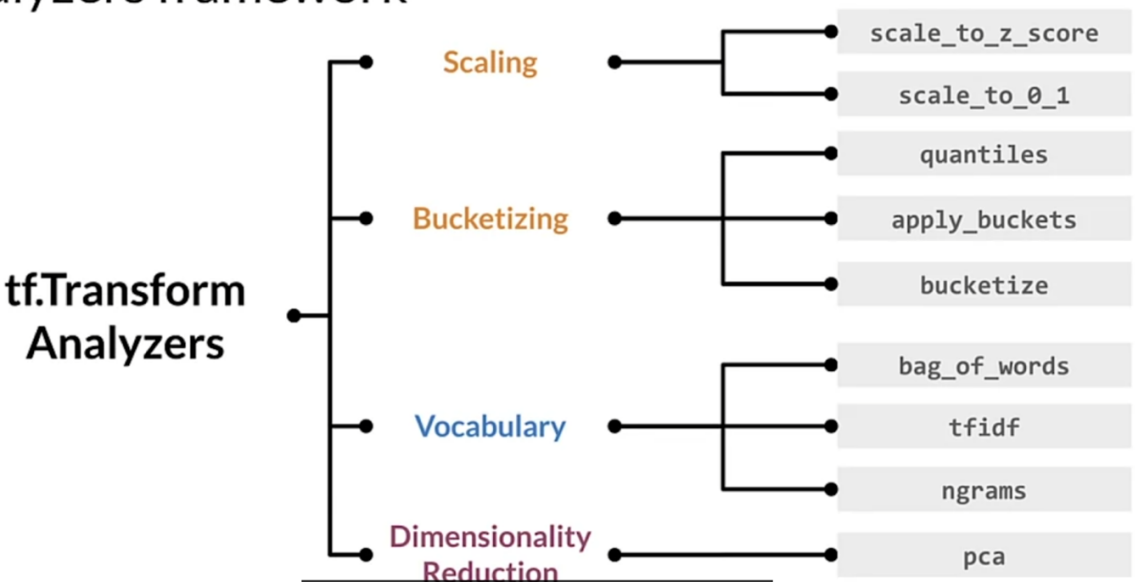


Transform is going to take the schema that was generated on the original split dataset, and it's going to do our feature engineering. Transform is where the feature engineering happen. That gets given the Trainer, there's Evaluator that evaluates the results, a set of Pusher that pushes it to our deployment targets which is, however we're serving our models, so TENSORFLOW SERVING, or JS, or LITE, wherever it is that we're serving our model. Internally, if we want to look at the transform component, it's getting inputs from, as we saw, 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 than can be really inferred by SchemaGen. That's referred to as curating the schema. Transform gets that and it also gets 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.

Benefits of using tf.Transform

- Emitted tf.Graph holds all necessary constants and transformations
- Focus on data preprocessing only at training time
- Works in-line during both training and serving
- No need for preprocessing code at serving time
- Consistently applied transformations irrespective of deployment platform

Analyzers framework



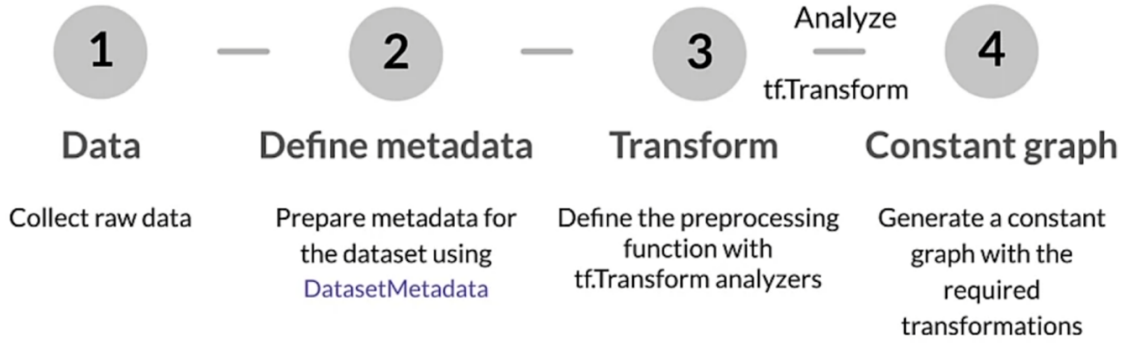
tf.Transform preprocessing_fn

```
def preprocessing_fn(inputs):
    ...

for key in DENSE_FLOAT_FEATURE_KEYS:
    outputs[key] = tft.scale_to_z_score(inputs[key])

for key in VOCAB_FEATURE_KEYS:
    outputs[key] = tft.vocabulary(inputs[key], vocab_filename=key)

for key in BUCKET_FEATURE_KEYS:
   outputs[key] = tft.bucketize(inputs[key], FEATURE_BUCKET_COUNT)
```

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
```

Hello world with tf.Transform



Collect raw samples (Data)

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

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)
    }))
```

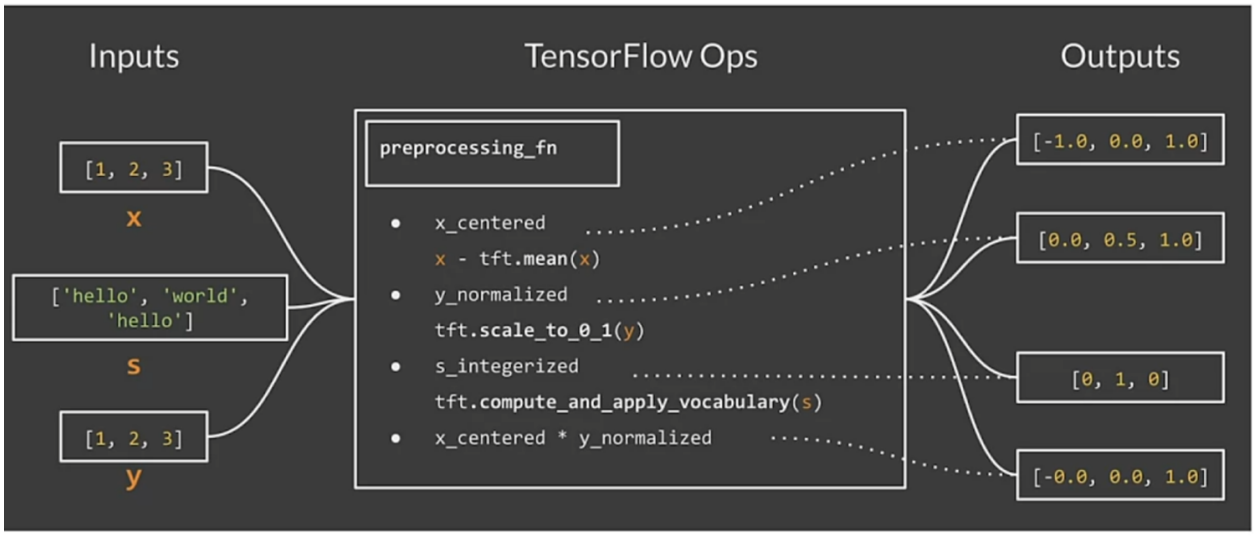
Preprocessing data (Transform)

```
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)
```

Preprocessing data (Transform)

```
    return {
      'x_centered': x_centered,
     'y_normalized': y_normalized,
     's_integerized': s_integerized,
     'x_centered_times_y_normalized': x_centered_times_y_normalized,
    }
```

Tensors in… tensors out



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))
```

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()
```

Before transforming with tf.Transform

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

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,
```

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

Outline

- Introduction to Feature Spaces
- Introduction to Feature Selection
- Filter Methods
- Wrapper Methods
- Embedded Methods

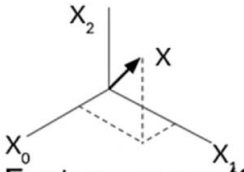
what are feature spaces? Well, a feature space is defined by the n dimensional space that your features defined. So if you have two features, a feature space is two dimensional. If you have three features, its three dimensional and so forth, it does not include the target label. So that we're just talking about your features. So if this is your feature factor, you have future vector X. And it has values from zero to D. Here. That's the dimensionality of that vector. And that defines a space. So in this case that's a three D space. We have three features. We have a 3D. Space, or if we're looking at it in a 2D space with with a two dimensional feature vector, we could express it as a scatter plot with a 2D space. Those are fairly easy for us as humans to imagine.

Feature space

- N dimensional space defined by your N features
- Not including the target label

$$X = \begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_d \end{bmatrix}$$

Feature vector



Feature space (3D)

Scatter plot (2D)

feature vector, we could express it as a scatter plot with a 2D space.

Feature space

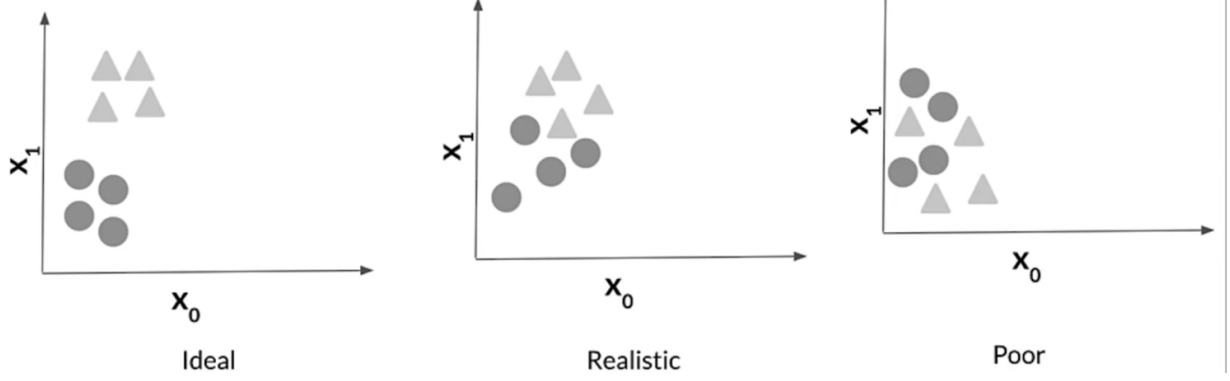
3D Feature Space

No. of Rooms X_0	Area X_1	Locality X_2	Price Y
5	1200 sq. ft	New York	$40,000
6	1800 sq. ft	Texas	$30,000

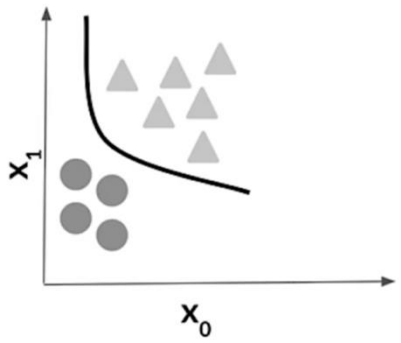
$Y = f(X_0, X_1, X_2)$

f is your ML model acting on feature space X_0, X_1, X_2

2D Feature space - Classification



Drawing decision boundary



Model learns decision boundary

Boundary used to classify data points

Feature space coverage

- Train/Eval datasets representative of the serving dataset
  - Same numerical ranges
  - Same classes
  - Similar characteristics for image data
  - Similar vocabulary, syntax, and semantics for NLP data

Ensure feature space coverage

- Data affected by: seasonality, trend, drift.
- Serving data: new values in features and labels.
- Continuous monitoring, key for success!

Feature selection

All Features

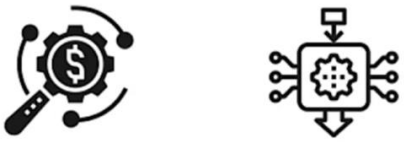
Feature selection

Useful features

- Identify features that best represent the relationship
- Remove features that don't influence the outcome
- Reduce the size of the feature space
- Reduce the resource requirements and model complexity

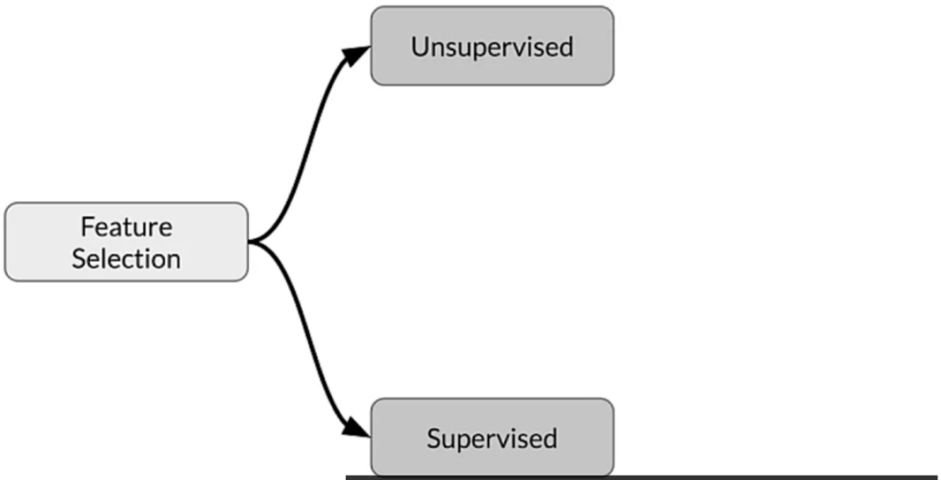
Why is feature selection needed?

Reduce storage and I/O requirements



Minimize training and inference costs

Feature selection methods



Unsupervised feature selection

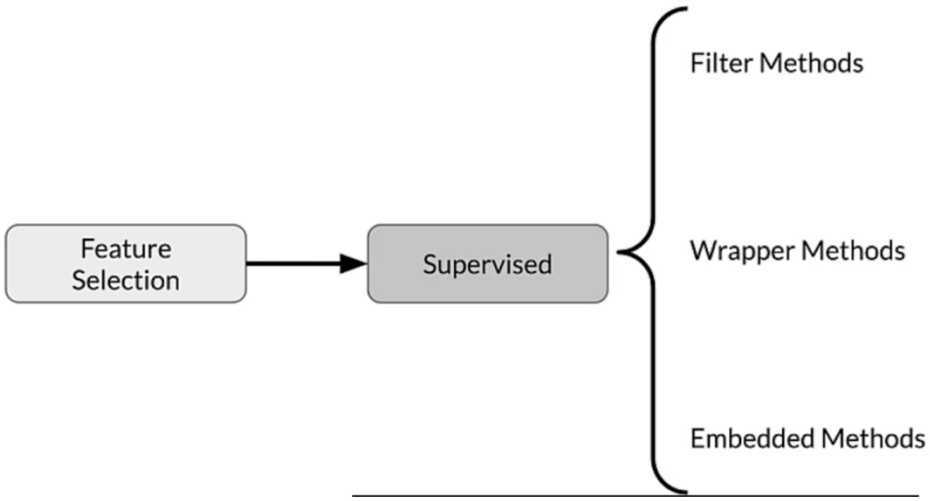
1. **Unsupervised**

- Features-target variable relationship not considered
- Removes redundant features (correlation)

Supervised feature selection

2. **Supervised**
   - Uses features-target variable relationship
   - Selects those contributing the most

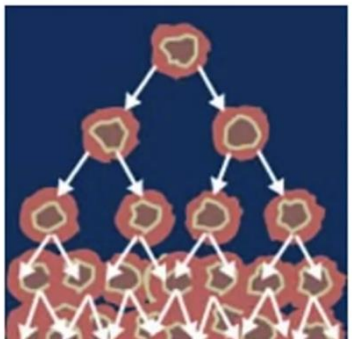
Supervised methods



Practical example

Feature selection techniques on Breast Cancer Dataset (Diagnostic)

Predicting whether tumour 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
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	[redacted]	[redacted]	[redacted]	0.4601	0.1189	NaN

Irrelevant features

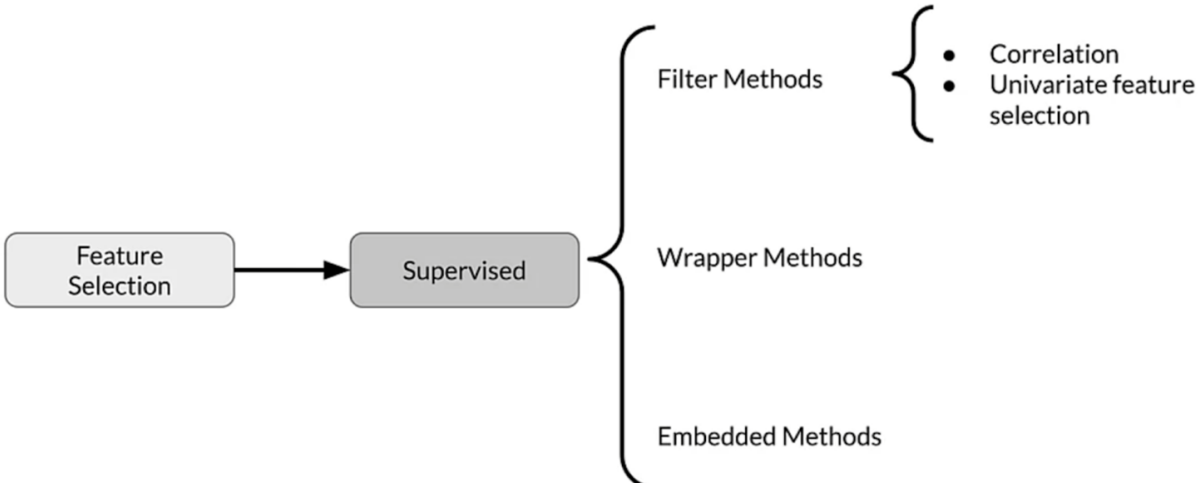
Performance evaluation

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

Filter methods



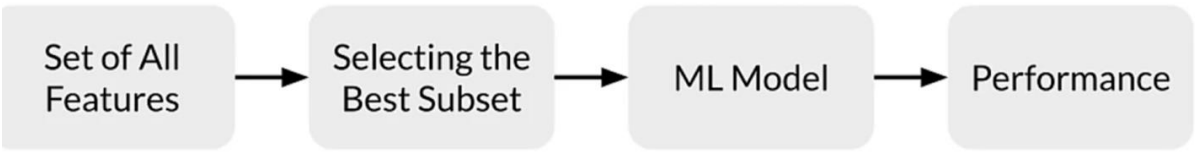
Filter methods

- Correlated features are usually redundant
  - Remove them!

Popular filter methods:

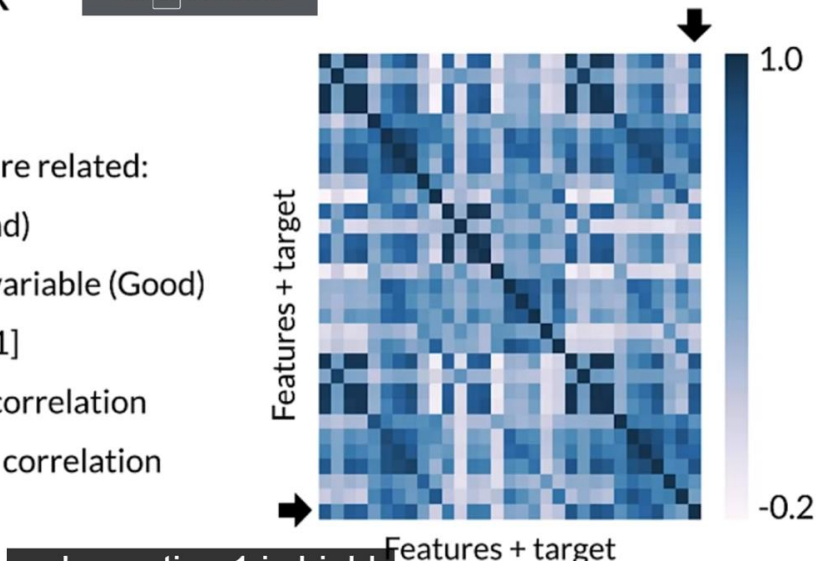
- Pearson Correlation
  - Between features, and between the features and the label
- Univariate Feature Selection

Filter methods



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



Feature comparison statistical tests

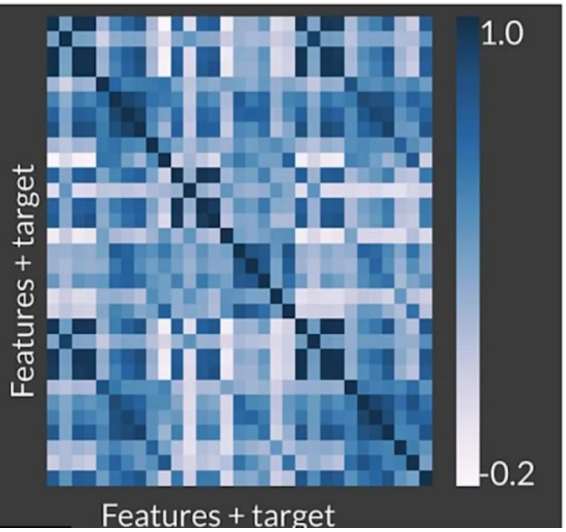
- Pearson's correlation: Linear relationships
- Kendall Tau Rank Correlation Coefficient: Monotonic relationships & small sample size
- Spearman's Rank Correlation Coefficient: Monotonic relationships

Other methods:

- Mutual information
- F-Test
- Chi-Squared test

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()
```



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]
```

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 feature selection in SKLearn

SKLearn Univariate feature selection routines:

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

Statistical tests available:

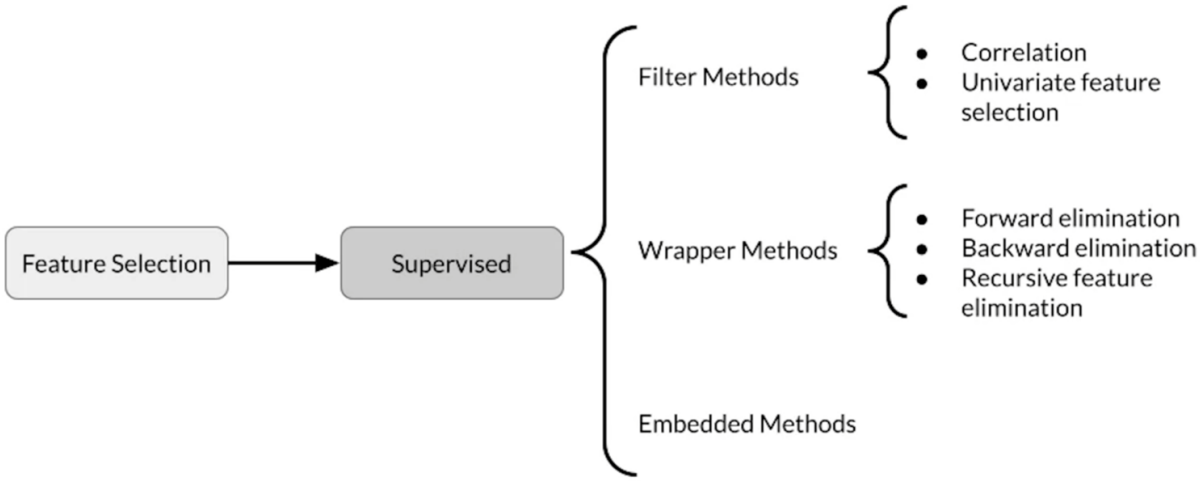
- Regression: `f_regression, mutual_info_regression`
- Classification: `chi2, f_classif, mutual_info_classif`

SelectKBest implementation

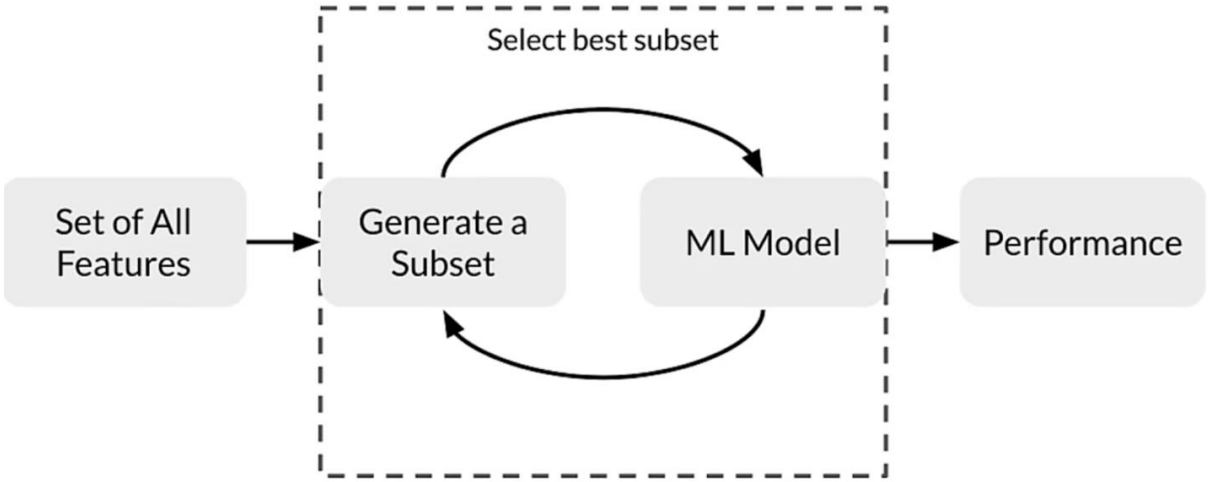
```
def univariate_selection():

  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
```

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**

Wrapper methods



Wrapper methods



Wrapper methods

Popular wrapper methods

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

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

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

Recursive feature elimination (RFE)

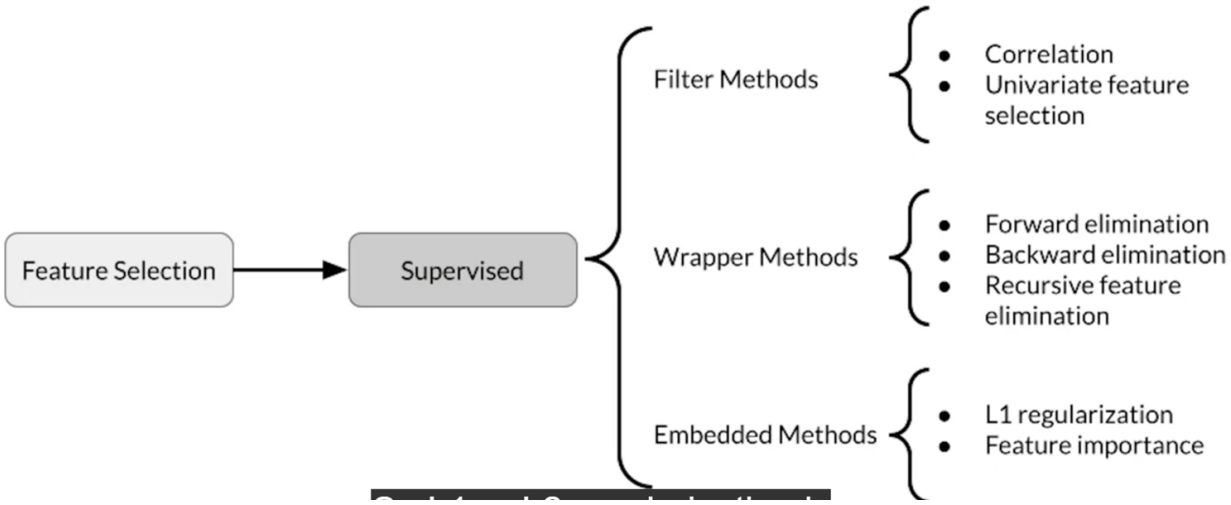
1. Select a model to use for evaluating feature importance
2. Select the desired number of features
3. Fit the model
4. Rank features by importance
5. Discard least important features
6. Repeat until the desired number of features remains

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)
```

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**

Embedded methods



Feature importance

- Assigns scores for each feature in data
- Discard features scored lower by 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.*

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
```

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
```

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)
```

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

Review

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