Google Cloud

Day 4

Machine Learning with TensorFlow on GCP

How Google does Machine Learning

Launching into ML

Introduction to TensorFlow

Feature Engineering

The Art and Science of ML

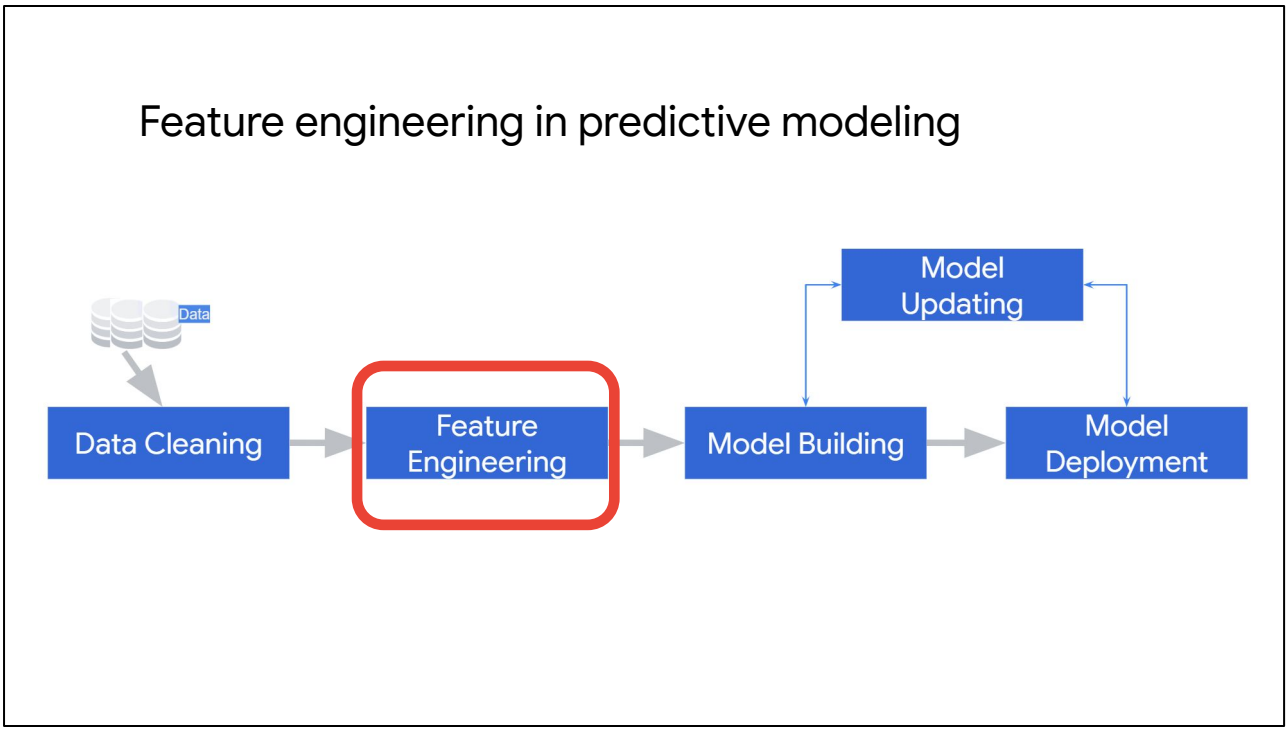
This is the fourth chapter of the *Machine Learning with TensorFlow on GC* specialization.

Welcome to Feature Engineering.

Agenda

- An Overview of Feature Engineering
- Raw Data and Features
- Feature Engineering
- Feature Crosses
- tf.transform

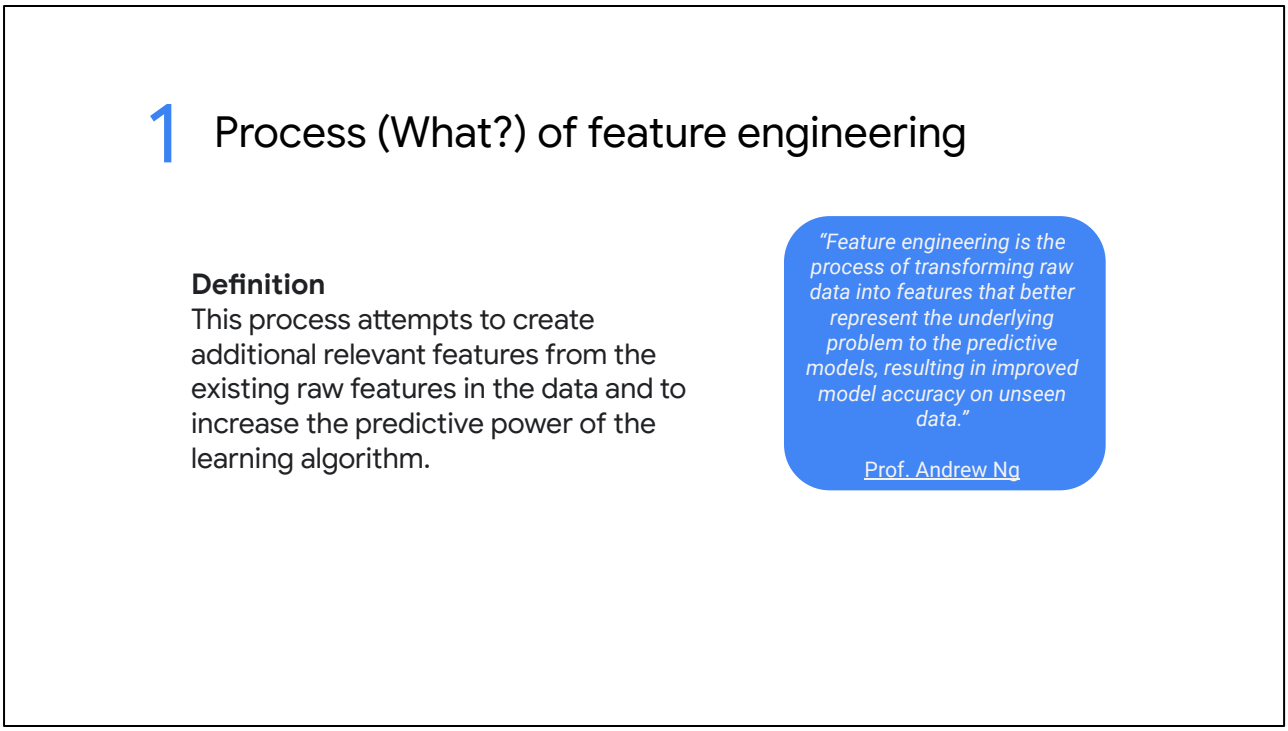
In this course we introduce feature engineering. We first start with an overview of feature engineering.
Next, we will discuss raw data and the importance of transforming them into features. We then look at how to perform basic to advanced feature engineering - which includes engineering feature crosses. Finally, we discuss tf.transform, which is a library for preprocessing data with TensorFlow. **tf**. **Transform** is useful for data that requires a full-pass, such as: (1) Normalize an input value by mean and standard deviation. (2) Convert strings to integers by generating a vocabulary over all input values.



Predictive models are constructed using supervised learning algorithms where classification or regression models are trained on historical data to predict future outcomes.

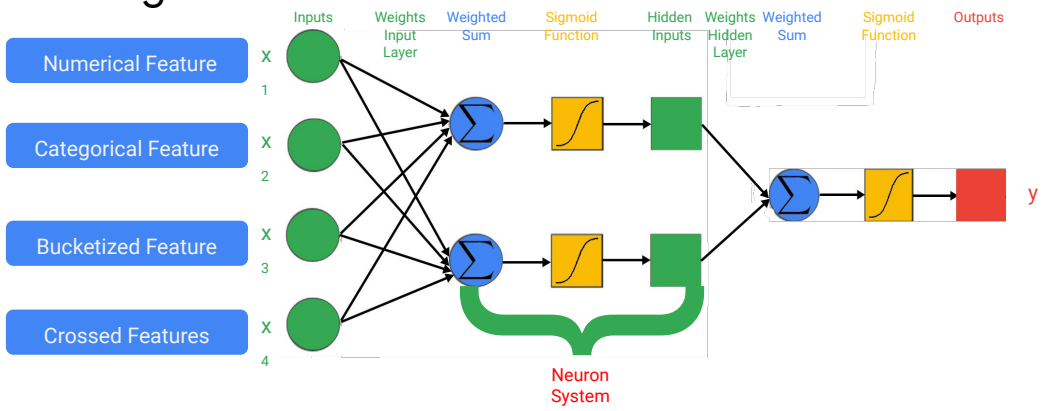
Feature engineering is a crucial step in the process of predictive modeling. It involves the transformation of a given feature, with the objective of reducing the modeling error for a given target.

The underlying representation of the data is crucial for the learning algorithm to work effectively. The training data used in machine learning can often be enhanced by extraction of features from the raw data collected. In most cases, appropriate transformation of data is an essential prerequisite step before model construction.



Definition (What)
Feature engineering can be defined as a process that attempts to create additional relevant features from the existing raw features in the data and to increase the predictive power of the learning algorithm. It can also be defined as the process of combining domain knowledge, intuition, and data science skillsets to create features that make the models train faster and provide more accurate predictions.

2 Purpose (Why): To improve the accuracy of models by increasing the predictive power of learning algorithms

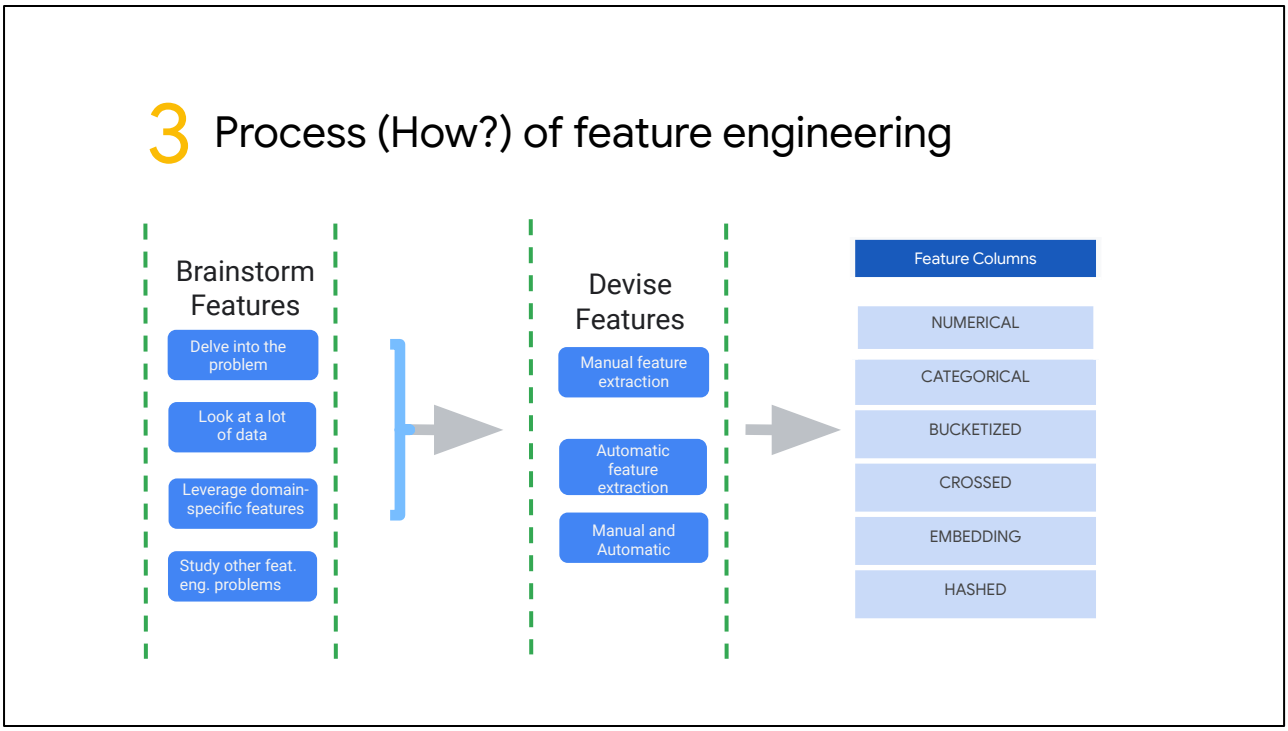


Machine learning models, such as neural networks, accept a feature vector and provide a prediction. These models learn in a supervised fashion where a set of feature vectors with expected output is provided.

From a practical perspective, many machine learning models must represent the features as real-numbered vectors because the feature values must be multiplied by the model weights. In some cases, the data is raw and must be transformed into feature vectors.

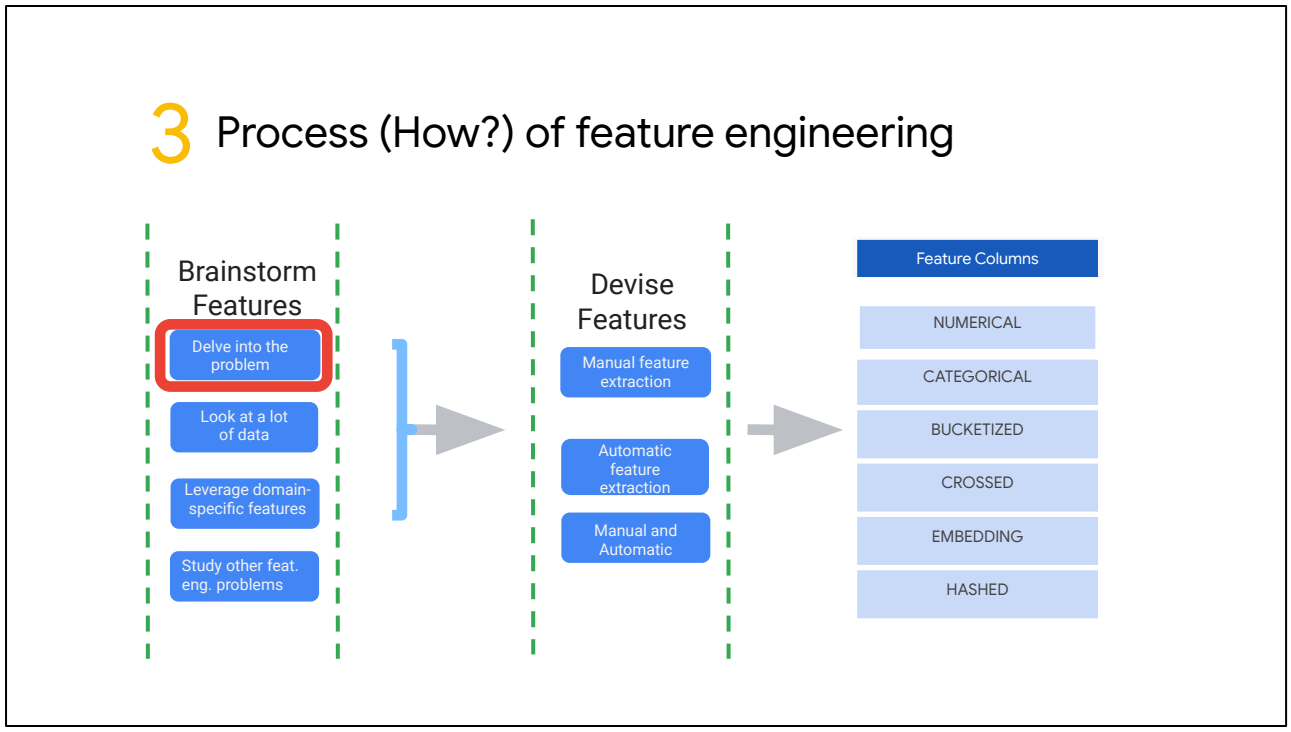
Features, the columns of your dataframe, are key in assisting machine learning models to learn. Better features result in faster training and more accurate predictions. As the diagram shows, feature columns are input into the model—not as raw data, but as feature columns.

Engineering new features from a provided feature set is a common practice. Such engineered features will either augment or replace portions of the existing feature vector. These engineered features are essentially calculated fields based on the values of the other features. As you will see later in the labs, feature vectors can be numerical, categorical, bucketized, crossed, and hashed.



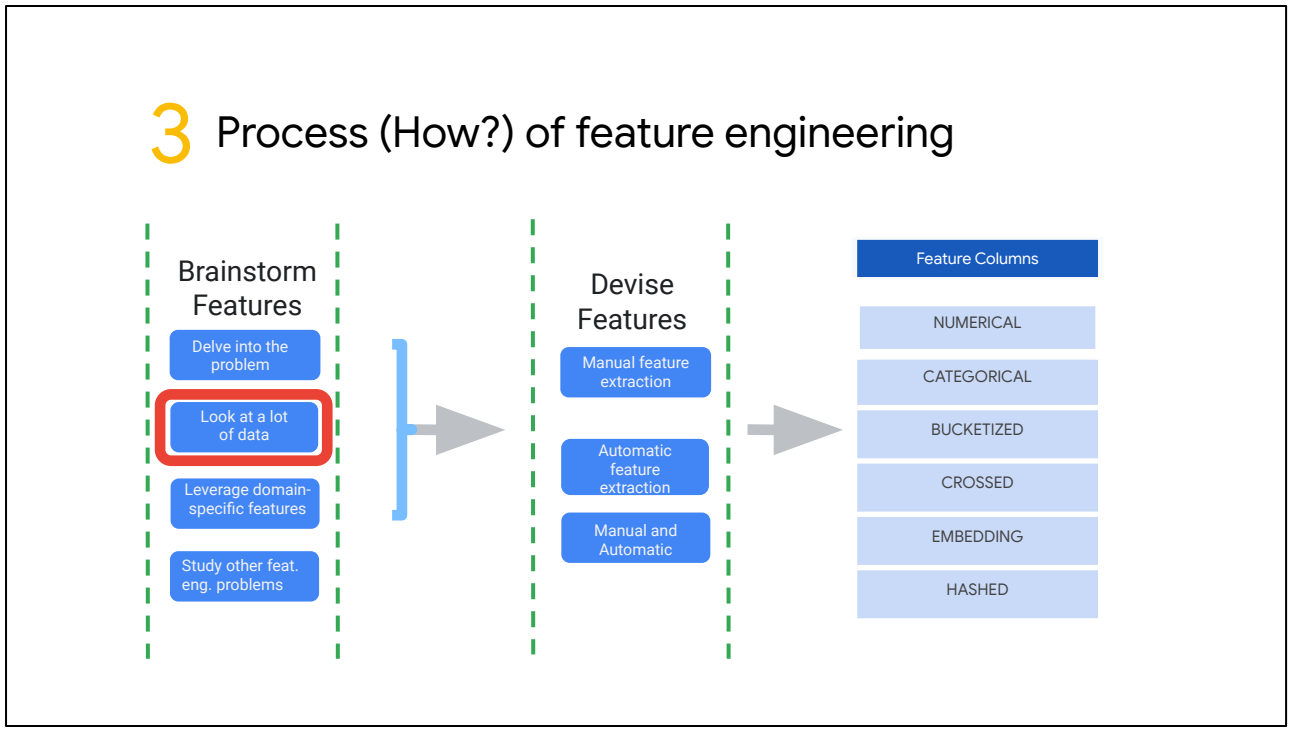
Engineering features is primarily a manual, time-consuming task.

The process involves brainstorming, where you



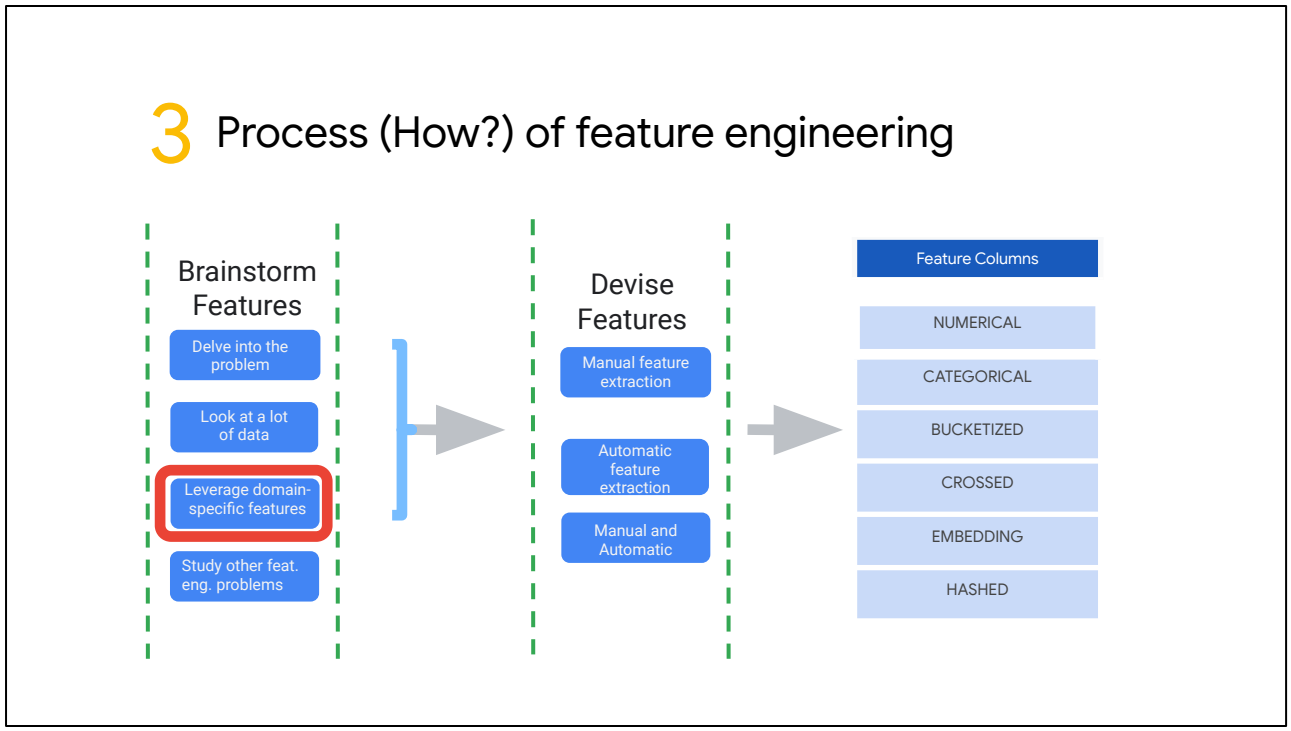
-- Delve into the problem. So, what does it mean to “delve into the problem?” If your goal is to solve a problem where you need to “predict” an outcome and get the best possible results from a predictive model, you need to determine if your source data can aide in this goal. In other words, how can you get the most out of your data for predictive modeling? This is the problem that the process and practice of feature engineering solves.

So you start by

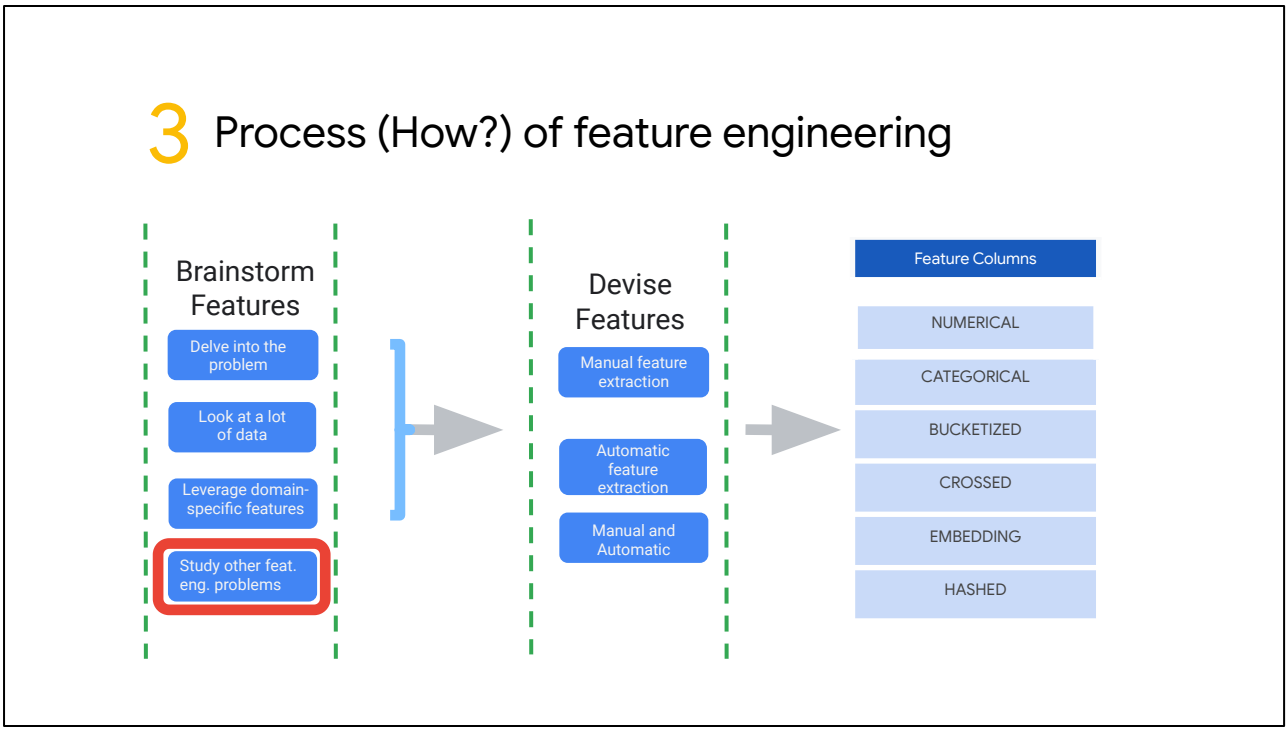


Looking at a lot of data. If you work as an academic researcher, you may have access to a plethora of data sources - from government data to industry data. If you work in a corporation, your data sources are the data your organization uses to manage the business. For example, you may be looking at customer data, sales data, product data, inventory data, operational data, and people management data.

And, this data could all be in different formats. The customer data could be a .csv file, the sales data could come from a .JSON file, the operational data could be in an .XML format...you get the picture. Depending on your problem and your domain, you then need to

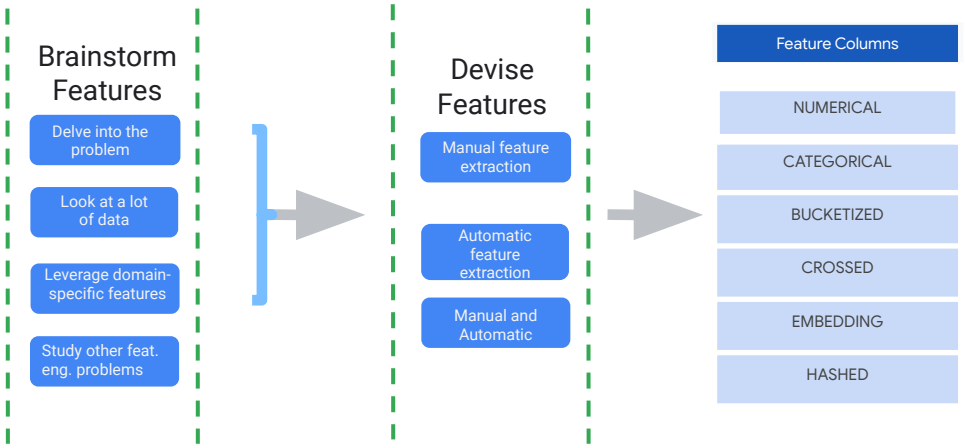


Leverage domain-specific engineered features. For example, if your machine learning problem is to predict seasonal sales based on past customer purchases in a geographic location, then you must either have domain specific knowledge or find domain specific features to solve the problem. In this case, you may be looking for data that allows you to create features for customer purchases at a specific date/time in a specific geo-location (or spatial) region.

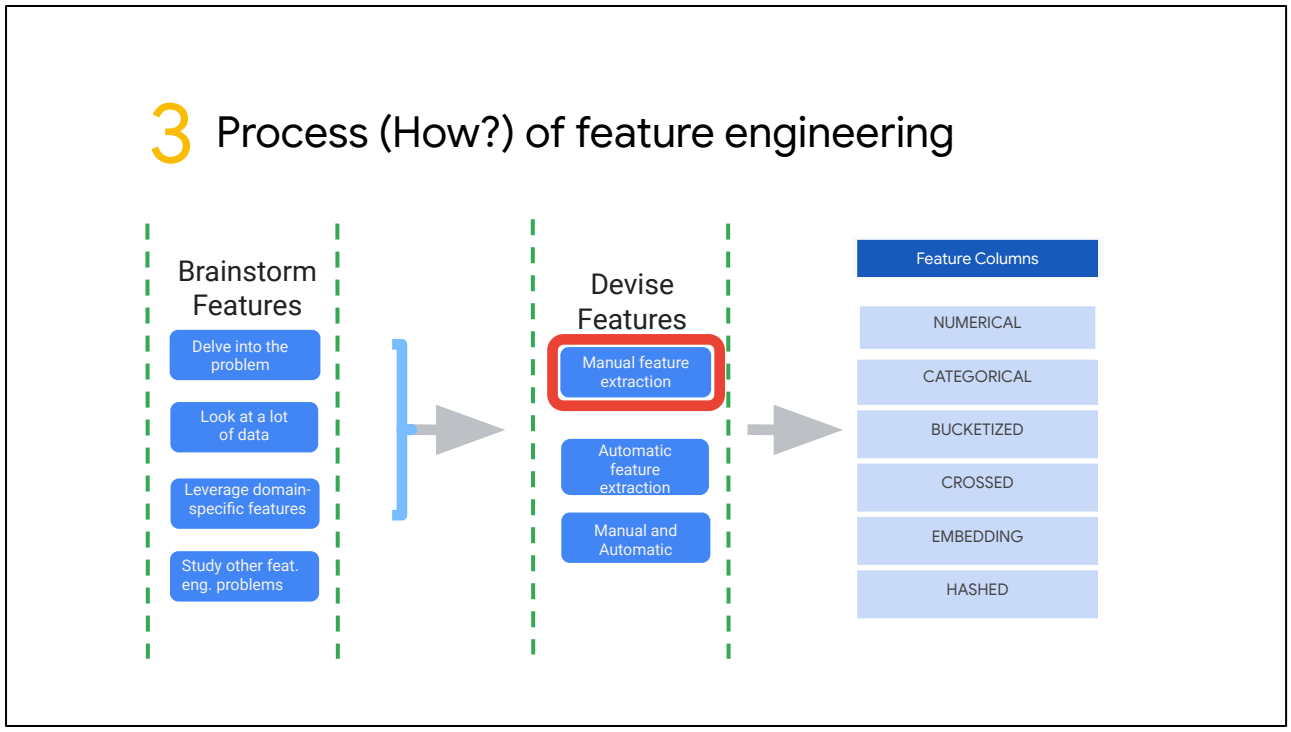


Studying other feature engineering problems is highly recommended. The example of seasonal sales, for example, lends itself to time series features and longitude and latitude features.

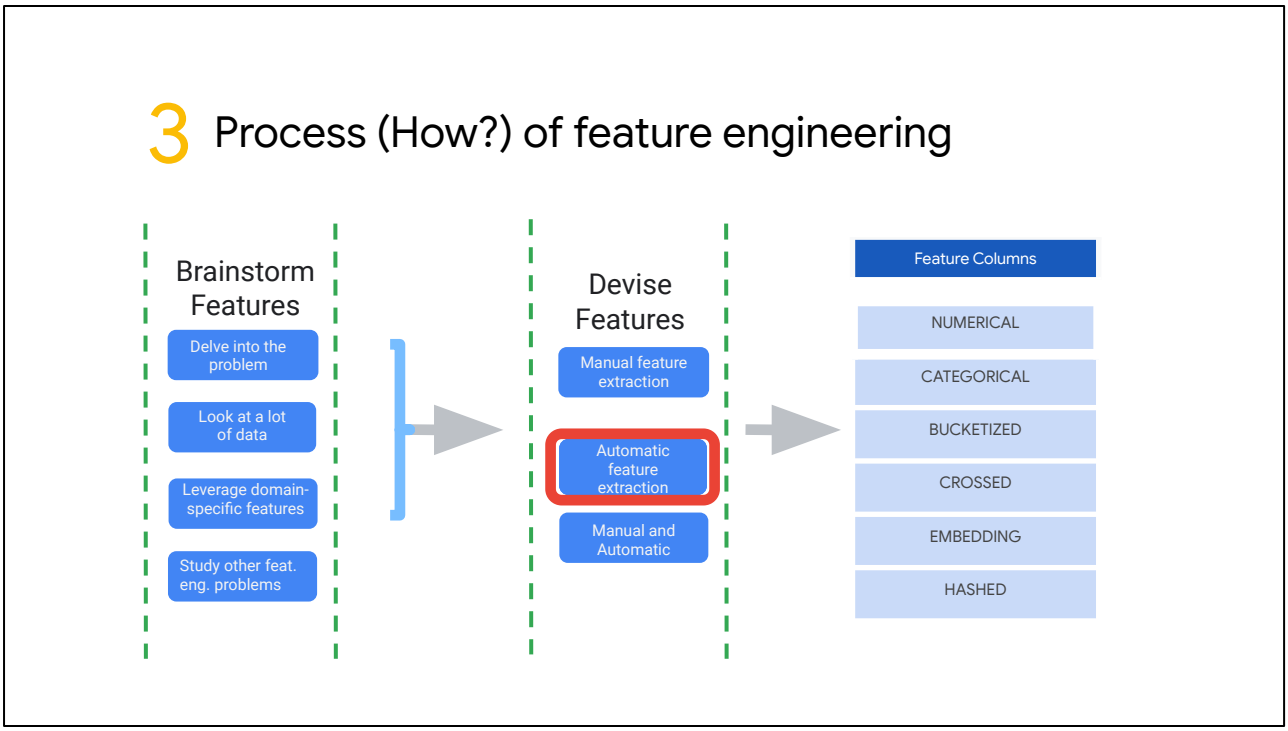
3 Process (How?) of feature engineering



You then must “devise” your features. This can be done in several ways.



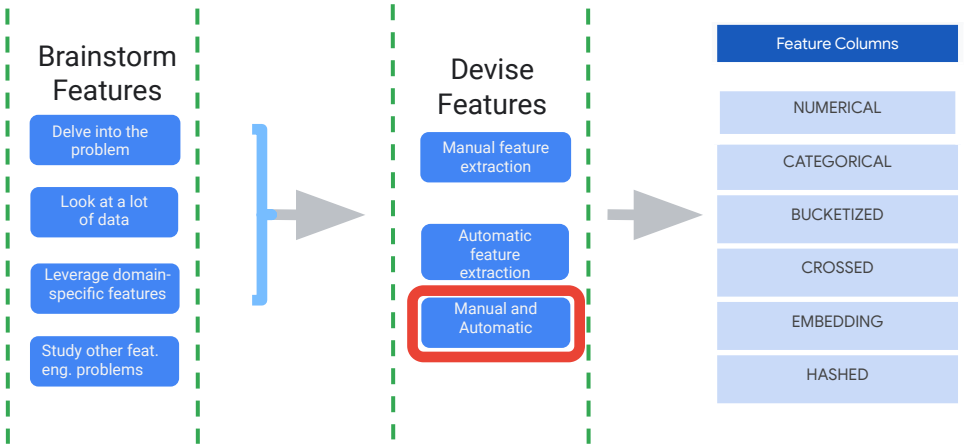
In manual feature extraction, features are constructed manually. In this module, we use manual feature extraction, where we manually devise features using Python and TensorFlow code libraries.



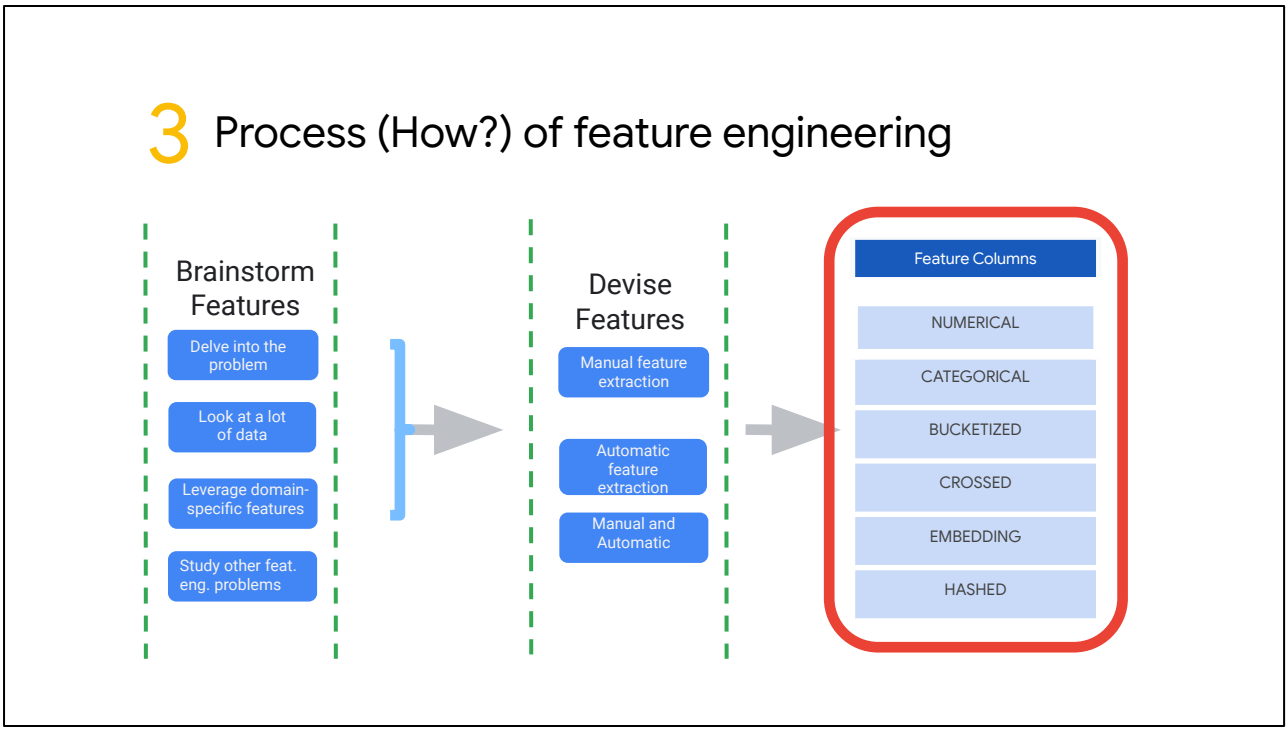
Feature extraction is a process of dimensionality reduction by which an initial set of raw data is reduced to more manageable groups for processing. For example, Principal Component Analysis (or PCA) is a way to reduce the number of variables or features in your dataset. As the name states, you are simply analyzing the principal components (your independent variables or features) to determine how well they predict your dependent variable or “target” (the final output you are trying to predict). In technical terms, you want to “reduce the dimension of your feature space.”

By reducing the dimension of your feature space, you have fewer relationships between variables to consider and you are less likely to overfit your model. In automatic feature extraction, the idea is to automatically learn a set of features from potentially noisy, raw data that can be useful in supervised learning tasks such as in computer vision.

3 Process (How?) of feature engineering



You can also devise features using a combination of both Automatic and Manual methods. As you can see

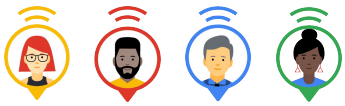


In this example here, the feature columns derived from this process can include: numerical, categorical bucketized, crossed, embedding, and hashed feature columns.

Different problems in the same domain may need different features

The key learning is that different problems in the same domain may need different features.

It depends on you and your subject matter expertise to determine which fields you want to start with for your hypothesis.

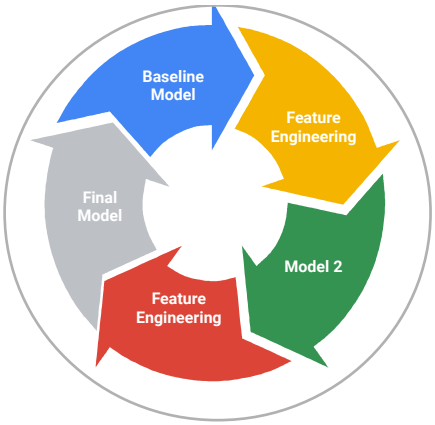


"Coming up with features is difficult, time-consuming, [and] requires expert knowledge. 'Applied machine learning' is basically feature engineering."
Andrew Ng

There is no well-defined basis for performing effective feature engineering. It involves domain knowledge, intuition, and most of all, a lengthy process of trial and error. The key learning is that different problems in the same domain may need different features. It depends on you and your subject matter expertise to determine which fields you want to start with for your hypothesis or problem.

4 Iterative Process

Example: Process can continue until RMSE is lowest



Recall that feature engineering can be defined as the process of transforming raw data into features that are input to the final model. However, feature engineering in reality is an iterative process. For example, you can improve the accuracy of models by increasing the predictive power of learning algorithms through iteration; in other words, you create a baseline model with little to no feature engineering (as determined by your data types), and then add feature engineering to see how your model improves. ~~In our BigQuery Machine Learning labs, you will see this iterative approach in action.~~

5 Feature engineering types

Type	Example
Using indicator variables to isolate key information.	Isolates a specific area for our training dataset.
Highlighting interactions between two or more features.	Sum of two features, product of two features, etc.
Representing the same feature in a different way.	Create a new feature “grade” with "Elementary School," "Middle School," and "High School" as classes.
	Group similar classes, and then group the remaining ones into a single "Other" class.
	Transform categorical features into dummy variables.

Feature engineering types include:

1. Using indicator variables to isolate key information: For example, geolocation features for New York city taxi service area isolates a specific area for our training dataset.

5 Feature engineering types

Type	Example
Using indicator variables to isolate key information.	Isolates a specific area for our training dataset.
Highlighting interactions between two or more features.	Sum of two features, product of two features, etc.
Representing the same feature in a different way.	Create a new feature “grade” with "Elementary School," "Middle School," and "High School" as classes.
	Group similar classes, and then group the remaining ones into a single "Other" class.
	Transform categorical features into dummy variables.

Highlighting interactions between two or more features: Interactions include the sum of two features, the difference between two features, the product of two features, and the quotient of two features.

5 Feature engineering types

Type	Example
Using indicator variables to isolate key information.	Isolates a specific area for our training dataset.
Highlighting interactions between two or more features.	Sum of two features, product of two features, etc.
Representing the same feature in a different way.	Create a new feature “grade” with "Elementary School," "Middle School," and "High School" as classes.
	Group similar classes, and then group the remaining ones into a single "Other" class.
	Transform categorical features into dummy variables.

Representing the same feature in a different way: For example, in numeric to categorical mappings, where you have “grade,” you can create categories and create a new feature “grade” with "Elementary School," "Middle School," and "High School" as classes.

5 Feature engineering types

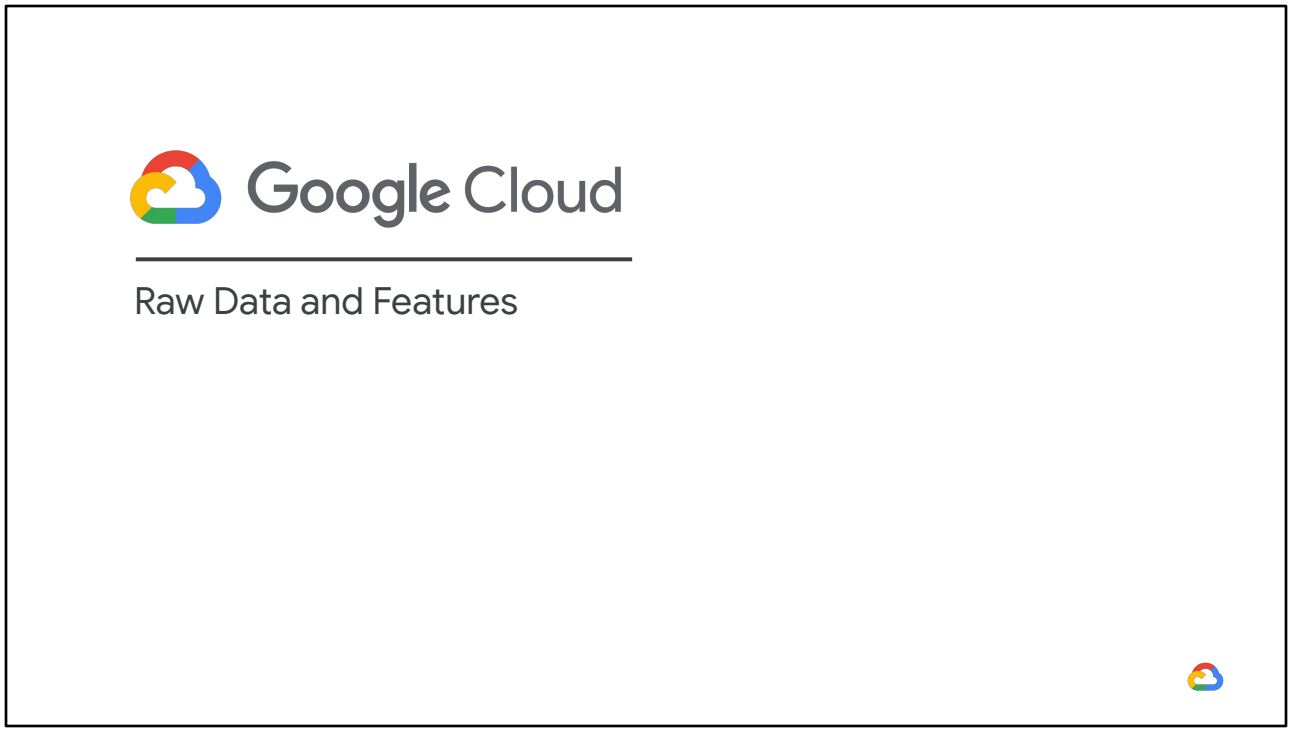
Type	Example
Using indicator variables to isolate key information.	Isolates a specific area for our training dataset.
Highlighting interactions between two or more features.	Sum of two features, product of two features, etc.
Representing the same feature in a different way.	Create a new feature “grade” with "Elementary School," "Middle School," and "High School" as classes.
	Group similar classes, and then group the remaining ones into a single "Other" class.
	Transform categorical features into dummy variables.

Another example is to group sparse classes, where you group similar classes and then group the remaining ones into a single “other” class.

5 Feature engineering types

Type	Example
Using indicator variables to isolate key information.	Isolates a specific area for our training dataset.
Highlighting interactions between two or more features.	Sum of two features, product of two features, etc.
Representing the same feature in a different way.	Create a new feature “grade” with "Elementary School," "Middle School," and "High School" as classes.
	Group similar classes, and then group the remaining ones into a single "Other" class.
	Transform categorical features into dummy variables.

And another example is to transform categorical features into dummy variables.



Welcome to Raw Data and Features.

Agenda

Turn raw data to features

Compare good versus bad features

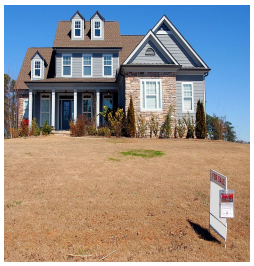
Represent features

Let’s start first with how you can turn your raw data to useful feature vectors that can then be used properly in your ML models

What raw data do we need to collect to predict the price of a house?

Lot size
Number of rooms

Historical sale price



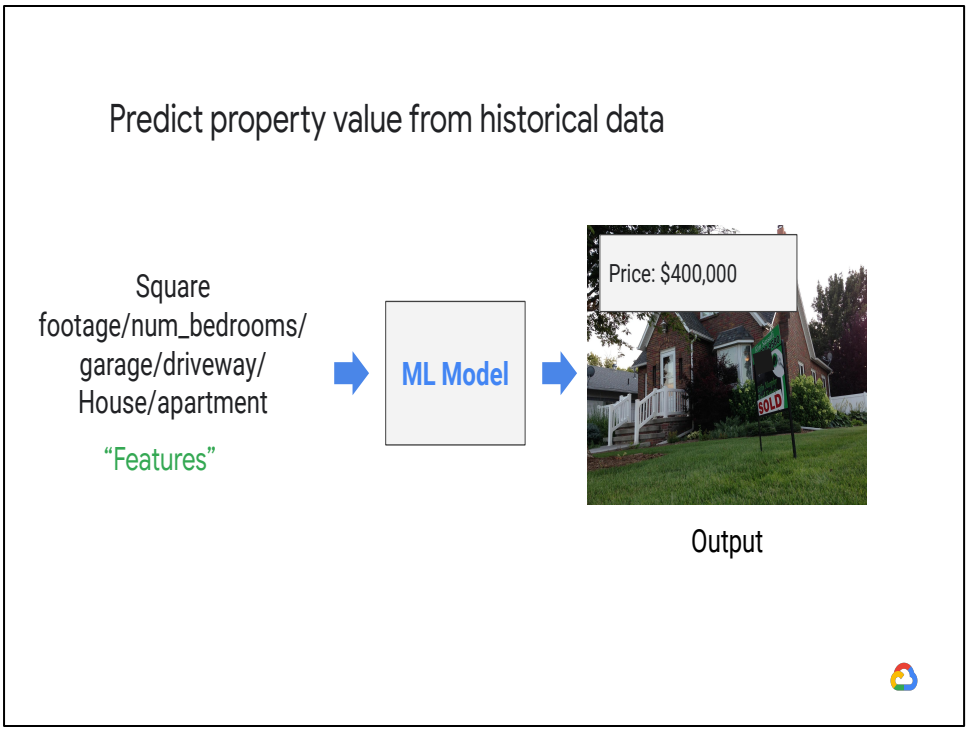
Location, location, location

Let’s say our end objective is to build a model to predict the price of a house for a given set of inputs. What would you first want to know about the house?

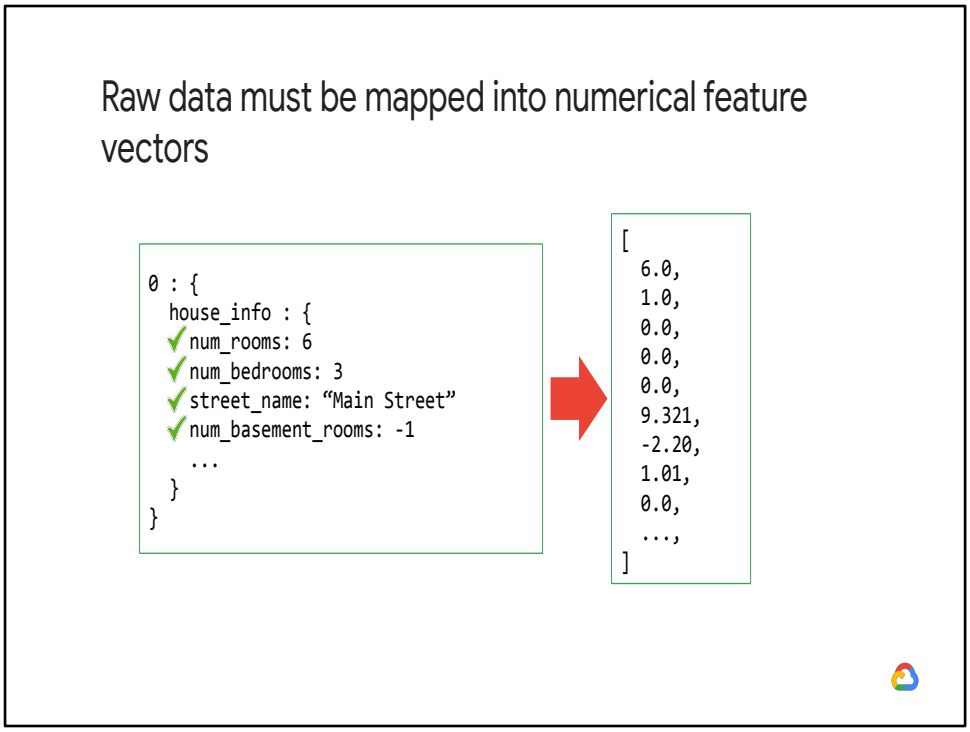
You might have said the square footage of the lot and house. Maybe the number of rooms.

If it was sold in the past, what was it sold for?

And you’ve probably already guessed that location, location, location could be a prime influencer of housing prices.



You must first choose your "features," that is, the data you will be basing your predictions on. Why not try to build a model that predicts the price based on the area of a house or apartment? Your features will be 1) the square footage, 2) number of bedrooms, 3) garage, 4) driveway, and 5) the category: "house" or "apartment."



Wouldn’t it be great if your raw data was already clean, and just the key fields you needed, as well as in a format you can just pass off to your ML model for training? Well this is almost never the case. Good feature engineering can take, on average, 50 to 75% of your time working with machine learning. It’s critical you get it right.

What you ultimately want to do is shown in this example. There’s raw data on houses on the left in the vector, which is to be mapped to one or more fields in the proto on the right, which you can then use in your ML model training. This might look like an easy mapping exercise to you.

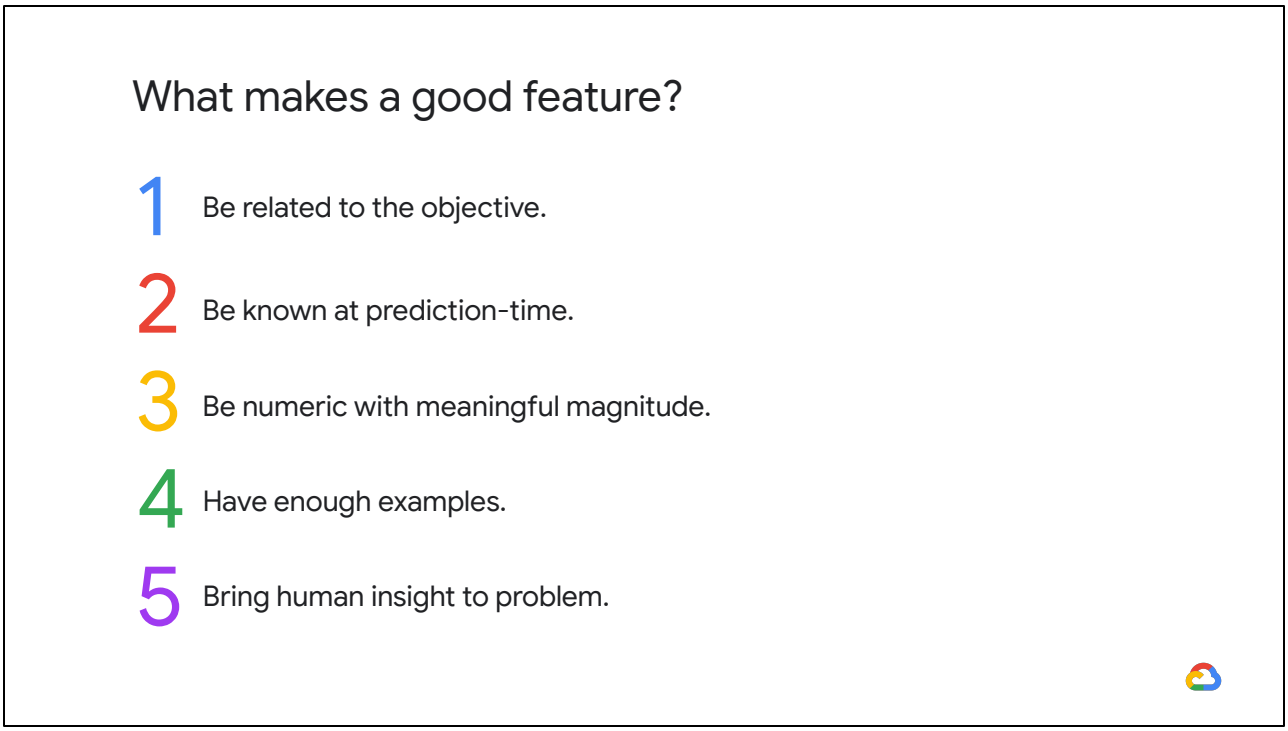
But how do you even know what a good feature is in the first place?

Agenda

Turn raw data to features

Compare good versus bad features

Represent features



Now, what makes a good feature, right? You want to basically take your raw data, and you want to represent it in a form that's amenable to machine learning.

So it has to be related to the objective, you don't want to just throw random data in there. That just makes the ML problem harder and the idea is to make the ML problem easier, right, make it easier for you to find the solution. If something is not related, throw it away.

You have to make sure that it's known at **prediction-time**. This can be surprisingly tricky.

You need to make sure it is **numeric.**

It has to have **enough examples**.

And you need to have some **human insights**.

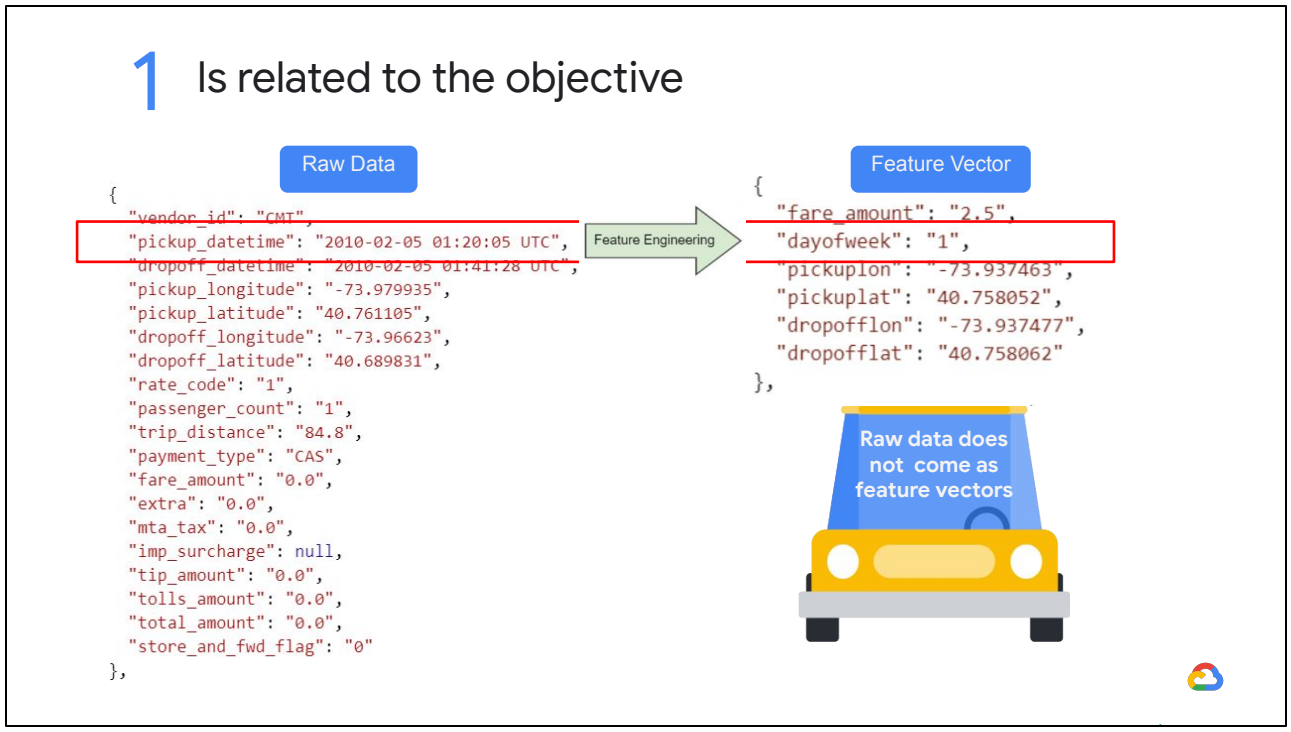
Be related to the objective

A good feature needs to be **related to what you're predicting**. You need to have some kind of a reasonable hypothesis of why a particular feature might matter for this particular problem. Don't just throw arbitrary data in there in the hope that there is some relationship somewhere. You don't want to do what's called data dredging. You don't want to dredge the large data set and find whatever spurious correlations exist. Because the larger the data set is, the more likely it is that there are lots of these spurious correlations.

In this example, just because we have a data point on whether chairs exist on the porch in the house photo or how many concrete blocks make up the driveway doesn't mean they should be included in your housing model.

You should have some reasonable idea of why these things, the feature value and the outcome. You know the outcome, basically the thing that's represented by the label. You have to have some reasonable idea of why they could be related in some way.

Now, you might be thinking that if you could tell if the driveway had cracks in it from the photo that would be a good feature -- keep that in mind when for later. We'll come back to it.



Related to the objective:

A good feature must be **related to what you're predicting**. You need to have a reasonable hypothesis about why a particular feature might matter for this particular problem.

In this example, of all the raw features on the left, we choose only six features for our machine learning problem. The label (or the value we are predicting) is the fare amount.

Quiz

Are these features related to the objective or not?

Test your knowledge with a quick quiz. You'll be shown some features and give you an objective and you say whether or not the features are related to that objective.

Predict the total number of customers who will use a certain discount coupon

1. Font of the text with which the discount is advertised on partner websites.
2. Price of the item the coupon applies to.
3. Number of items in stock.

Assume that you want to predict the total number of customers who will use a discount coupon. Which of these are related.

The **font of the text** in which the discount is advertised, yes or no? Absolutely. The bigger the font the more likely it is to be seen,perhaps. There is probably a difference between Comic Sans and Times New Roman. Some fonts are more trustworthy than others. The font of the text in which the discount is advertised. Yes, it's probably a good feature.

What about the **price of the item** the coupon applies to? You can imagine that people use a coupon more if the item costs less. So this could feature. But notice what you're doing. You're not just saying yes or no for a particular feature. You're saying "yes" because people may use a coupon more if the item is not highly priced. You're saying "Yes" people might use a coupon more if they get to see it and you get to see it better if the font is bigger. You need to have a reasonable hypothesis for why a particular feature might be good.

Number of items in stock. No, how would the user even know that. Out of stock versus in-stock could be a feature. But not 800 items versus thousand items.

Predict whether a credit card transaction is fraudulent or not

1. Whether the cardholder has purchased these items at this store before.
2. Credit card chip reader speed.
3. Category of item being purchased.
4. Expiry date of credit card.

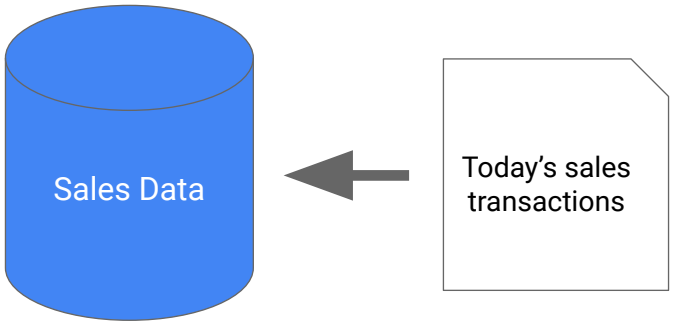
Consider this example. Predict whether the credit card transaction is fraudulent or not.

Whether the card holder has purchased these items at the store before, is that a good feature or is it not? It could be a feature. Is this a common purchase for this user or is it a completely unfamiliar unlikely thing? Whether a card holder has purchased items at the store before is probably a good feature.

Would you consider category of the item being purchased to be a good feature? There probably is some fraud committed with an item like a television and not as much fraud committed with a clothing item. The category of the item could be a signal, a feature that you want to use in your model.

What about the expiry date of the credit card would probably not be considered, perhaps however, the issue date would be considered, as new credit cards experience more fraud.

Prediction-time: Some data could be known immediately, and some other data is not known in real time



You would need to know the value at the time that you're predicting. Remember, the whole reason to build the machine learning model is so you can predict with it. If you can't predict with it, there is no point in building the machine learning model.
A common mistake people make, is to look into their data warehouse and take all the data in there, take all the related fields and throw it into the model. If you take all these fields and you use it in the machine learning model what happens when you're going to go predict with it?

When you predict with it, maybe you'll discover that your data warehouse had sales data. So that's an input into your model. How many things were sold the previous day would be an input into your model. But then it turns out that daily sales data *actually comes in a month later.* It takes some time for information to come out from your store. There is a delay in collecting this data. Your data warehouse has the information because somebody went through the trouble of taking all the data or joining the tables, putting them in there. **But at prediction time at real time, you don't have it.**

Some of the information in a data warehouse is known immediately, and some other information is not known in real time.
If you have used the data that is not known at prediction time as an input to your model, you don't have a numeric value for that input that your model needs.

You need to ensure that you will have every input and every feature, at prediction time. You want to ensure that those input variables are available. You're collecting it in

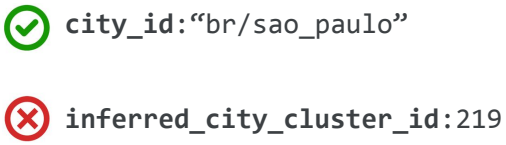
a timely manner.

In many cases you also have to worry about whether it's legal or ethical to collect that data at the time that you're doing the prediction. Sometimes that's information that you will have available to you in your data warehouse but you cannot collect it from the user at the time that you're trying to do the prediction.
If that's the case then you cannot use it in your ML model.

You cannot feed your model what you don't know at prediction-time

Earlier you used the housing price prediction model. If you had today's sale price of the house in the model in the dataset, the model could just output what that price was and be perfectly accurate during training because it had access to this magic data field. But come prediction time, your new houses for sale won't have already been sold so your model is useless because you cannot feed it what you do not know at prediction.

What's wrong with the second feature?



In this example, consider why the second is a bad feature? What could go wrong? What would happen if the cluster ID was taken from another model? What if that model updates without telling you? Will you still be able to learn anything from your training data?

Feature definitions should not change over time.

Quiz

Is the value knowable at prediction time or not?

Time for another quiz. Are the features we're going to show you knowable at prediction time or not?

Predict the total number of customers who will use a certain discount coupon

1. Total number of discountable items sold.
2. Number of discountable items sold the previous month.
3. Number of customers who viewed ads about item.

Taking the discount coupon case you looked at earlier, and the total number of discountable items that have been sold, how long a period are you looking at this total number for? How long does it take for you to know that value? This is not a yes or no answer, but it's a question that you need to ask before you take it as an input. The number of discountable items sold the previous month, this is getting closer. This seems like something that you should have available to you. So notice that there is a way to define this. If it's something as vague as total number of discountable items sold, it's too vague. You don't have a time period, you don't know how long it takes to collect.

If you make it more practical, for example, total number of discountable items sold the previous month. At this point, you've defined it in a way that you can have it. And of course, this time frame is going to depend on the latency in your system. So this is, again, a prompt for you to go find out these kinds of things. How long does it take for you to actually get this data before you can use this data in real time?

Number of customers who viewed ads about an item? Again, this is a question of timing. How long does it take for you to get the ad analysis back so you can use it?

You cannot train with current data and predict with stale data


```
SELECT name, COUNT(trans_id) AS count
FROM sales_warehouse
WHERE
     # filter out last three days
  trans_time <
  TIMESTAMP_SUB(
    CURRENT_TIMESTAMP(), INTERVAL 3 DAY
    )
```

If sales data at prediction time is only available with a three day lag, you should only train with sales data at least three days old

You cannot train with current data and predict with stale data. If you go to data warehouse for training you cannot use all of the values for a customer's credit card usage because not all of those values are going to be available at the same time.

What you would have to do is modify your training data to be as of three days ago. The key point is that you have to train with stale data, if stale data is what you will have in real time.

So let's think about this. Say you're doing a prediction on May 15, the data in your database is only going to be current as of May 12. Which means when you're training, and you're training on some data on Feb 12, that will use the input variable. You can only train with the number of times this card has been used as of Feb 9. You have to correspondingly correct for the staleness of your data.

If you've trained your model assuming that you know exactly the data up to the minute, but then at prediction time, you'll only know the data as of three days ago, then you'll have a very poorly performing machine learning model.

You have to think about the temporal nature of all of the input variables that you're using.

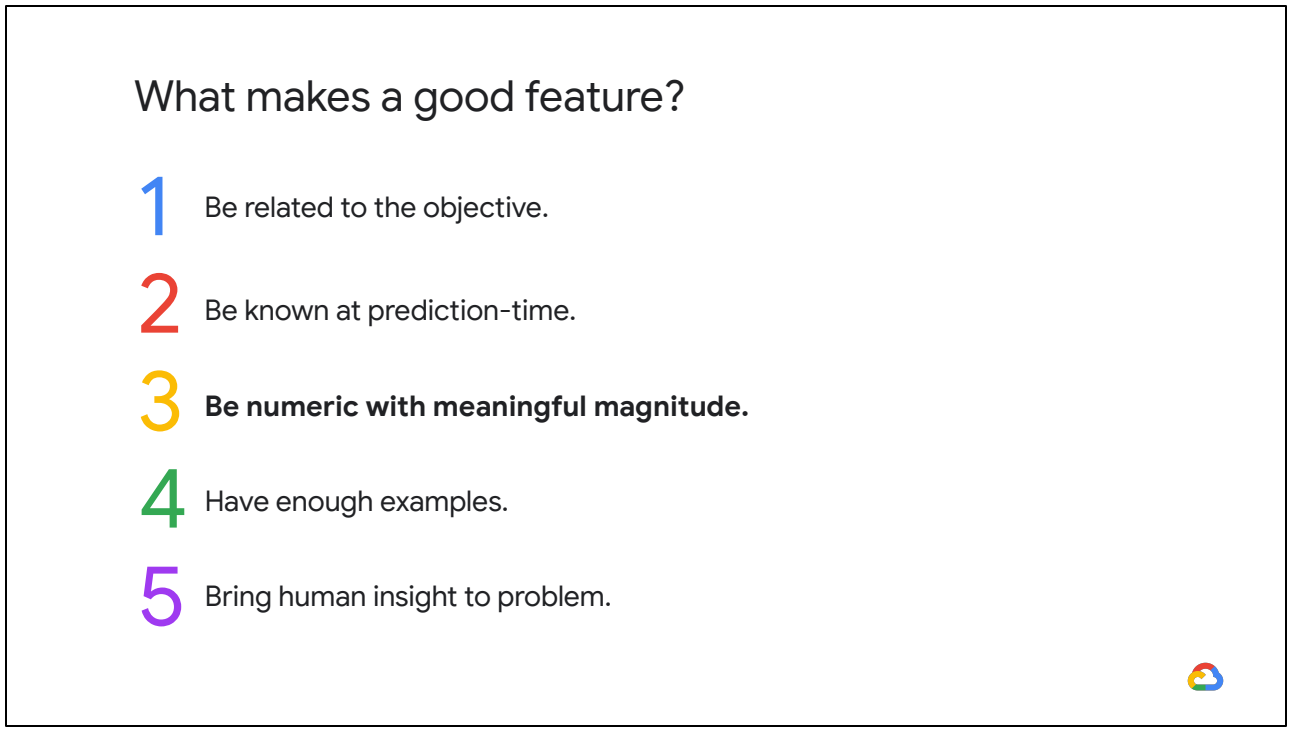
Predict whether a credit card transaction is fraudulent or not

1. Whether the cardholder has purchased these items at this store before.
2. Whether the item is new at the store (and can not have been purchased before).
3. Category of item being purchased.
4. Online or in-person purchase?

Okay, next one. Is the item new at the store? And so it, cannot have been purchased before. Yes, this is something you should know from your catalogue. Perfectly valid input.

Category of the item being purchased. No problem,again you'll know it. You'll know if this thing is a grocery item, or if this thing is an apparel item, or if this thing is an electronic item. No problem, we can look at the time, and we know what it is.

Online purchase or in-person purchase. Yes, you know this thing, too, in real time. It's not a problem.



Now, what makes a good feature, right? You want to basically take your raw data, and you want to represent it in a form that's amenable to machine learning.

So it has to be related to the objective, you don't want to just throw random data in there. That just makes the ML problem harder and the idea is to make the ML problem easier, right, make it easier for you to find the solution. If something is not related, throw it away.

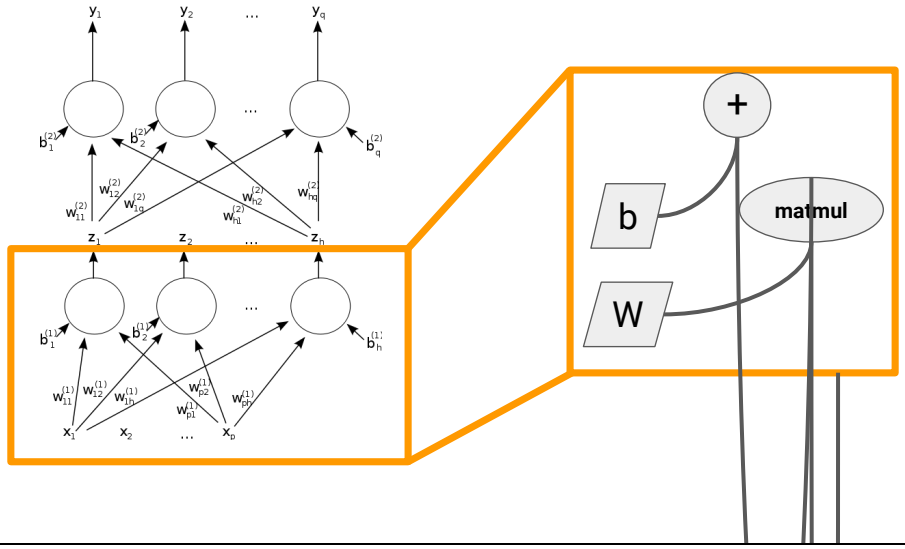
You have to make sure that it's known at **prediction-time**. This can be surprisingly tricky.

You need to make sure it is **numeric.**

It has to have **enough examples**.

And you need to have some **human insights**.

Neural networks are weighing and adding machines



The third key aspect of a good feature is that all your features have to be in numeric and they have to have a meaningful magnitude.

A neural network is simply an adding, multiplying, weighing machine. It's just carrying out arithmetic operations, computing trigonometric functions, algebraic functions on your input variables.

So your inputs better be numbers and those magnitudes better have some meaning so that a number two is, you know, in some sense twice number one. So what do we mean by this?

Quiz

Which of these are numeric with meaningful magnitude?

So let's take a quiz again. Which of these are numeric? Note: Non-numeric features can be used; it’s just that we need to find a way to represent them in numeric form.

Predict the total number of customers who will use a certain discount coupon

1. Percent value of the discount (e.g. 10% off, 20% off, etc.)
2. Size of the coupon (e.g. 4 cm^2, 24 cm^2, 48 cm^2, etc.)
3. Font an advertisement is in (Arial, Times New Roman, etc.)

You're trying to predict the number of coupons that are going to be used, and you're looking at different features of that discount coupon. Would the percent value of the discount, for example, 10% off, 20% off, etc, be numeric? It would. Meaningful magnitude is a 20% coupon worth twice a 10% off coupon. The percent value of the discount is a numeric input.

You're trying to predict the number of coupons that are going to be used, and you're looking at different features of that discount coupon.
So the percent value of the discount,(for example, 10% off, 20% off, etc,) Would you consider this to be numeric? It would. Meaningful magnitude is a 20% coupon worth twice a 10% off coupon. The percent value of the discount is a numeric input.
Consider the size of the coupon, if you define it as 4 square centimeters, 24 square centimeters, 48 squared centimeters, this is numeric.
Yes, so six times more visible than one that's 4 square centimeters? Sure, that makes sense, so you could imagine that this is numeric, however it's also unclear whether the magnitude are actually meaningful. So if this were an ad the size of a banner ad that you were placing, larger ads are better and you could argue that that makes sense. However, if it's a physical coupon, if it's something that goes out on paper, then you have to wonder whether a 48 square centimeter coupon is twice as good as a 24 square centimeter coupon.
If you were to you change this problem a little bit, if you define a size of coupon as small, medium and large. At that point, small, medium or large, are they numeric? Not at all.
It is not suggested that you cannot have categorical variables as inputs to neural

networks. You can, it's just that you can't just use small, medium or large directly, you have to do something smart to them.
You would have to find a way to represent them in numeric form. You will look at how to do this shortly.

If you look at the third point, font an advertisement is in (Arial 18, Times New Roman 24, etc.), would you say this is numeric? It isn’t, but how do you convert Times New Roman to numeric?
You could say Arial is number one, Times New Roman is number two, Roborto is number three, and Comic Sans is number four. That's a number code, they don't have meaningful magnitudes.
If you set Arial as one and Times New Roman as two, Times New Roman is not twice as good as Arial. So the meaningful magnitude part is important.

Predict the total number of customers who will use a certain discount coupon

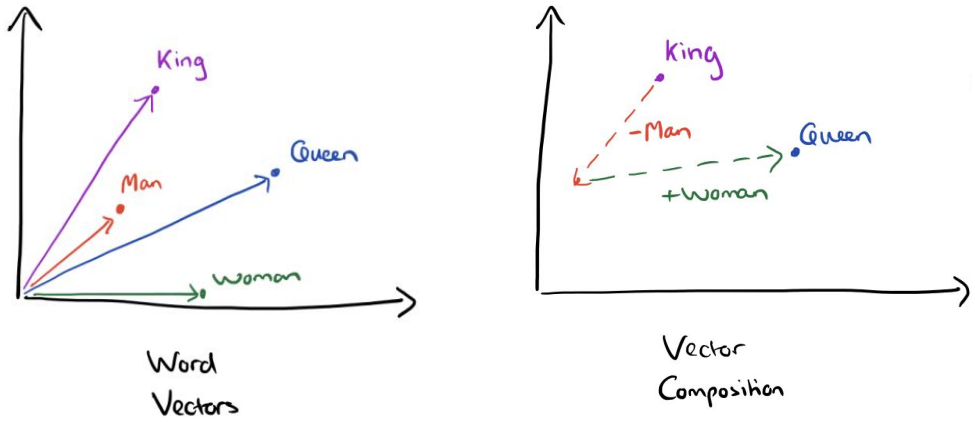
4 Color of coupon (red, black, blue, etc.)

5 Item category (1 for dairy, 2 for deli, 3 for canned goods, etc.)

Color of the coupon, red, black, blue, etc, again, no, these are not numeric. They don't have meaningful magnitudes. Again, we can come up with a number like RGB values to make colors numbers, but they're not going to be meaningful numerically. If I subtract two colors and the difference between them is three, does that mean if I subtract two other colors and the difference between them is also three, are these too commensurated? No, and that's a problem.

Item category (1 for diary, 2 for deli, 3 for canned goods, etc.) these are categorical, not numeric. It's not that you can't use non-numerical values, you need to do something to them, and you need to look at what things need to do be done to them. So as an example, suppose you have words in a natural language processing system, the things that you do to the words to make them numeric is that you run, typically, something called Word2vec.

Use Word2vec to make word vectors



Word2vec is a very standard technique. You take your words and apply this technique to make those words vectors, so each word becomes a vector.

When you look at these vectors at the end of Word2vec, these vectors are such that if you take the vector from man and you take the vector from woman and you subtract them, the difference that you get, it's going to be the similar difference to if you take the vector for king and you take the vector for queen and you subtract them.

That's what Word2vec does.

Changing an input variable that's not numeric to be numeric is not a simple matter, it’s requires a lot of work.

You could just go ahead and throw random encoding in there. But your ML model is not going to be as good as if you had started with a vector encoding that's nice enough that it understands the context of male and female, man and woman, king and queen.

You need to have numeric features, they need to have meaningful magnitudes. They need to be useful, you need to be able to do arithmetic on them, you need to find vector representations in such a way that these kinds of qualities exist.

Have enough examples of feature value in the dataset

You need to have enough examples of in feature value in the dataset. So it's the value of the feature, you need to have enough examples of this. Rule of thumb and this is just a rule of thumb, is that you need to have at least five examples of any value, before using it in your model. It's just rule of thumb. At least five examples of a value before you use it in training. So what is meant by this?

In the example on the left, consider the following. If you have purchasing category equals auto, then you must have enough transactions, fraud or no fraud of auto purchases, so you should have enough fraudulent auto transactions, enough not fraudulent auto transactions. If you have only three auto transactions in your dataset, and all three of them are not fraudulent, then essentially a model is going to learn that nobody can ever commit fraud on auto transactions. And that's gonna be a problem.

You want to avoid having values of which you don't have enough examples. For every value of a particular column you need to have at least five examples. Think back to the cracked driveway example, if you believe the photo showing a cracked driveway could be a good indicator of housing price, be sure to have enough examples for your model to train off of.

Quiz

Which of these will it be difficult to get enough examples?

Which of these features will it be difficult to get enough examples for?

Predict the total number of customers who will use a certain discount coupon

1. Percent discount of coupon (20%, 30%, etc.)
2. Date that promotional offer starts.
3. Number of customers who opened advertising email.

Consider again that you are trying to predict the number of customers who will use a discount coupon. And you have, as a feature, the percent discount of the coupon. So consider, in this example, that you have a coupon that has a 10% discount. You'll probably have at least five times that a 10% off coupon has been used. If you have a 10% off or a 5% off or a 15% off, you'd probably have at least five examples of these. But what if you have this one special customer to whom you give an 85% discount? Can you use that in your dataset?
No, you don't have enough samples. That 85% is now way too specific. You don't have enough examples of an 85% discount, so you have to throw it out, or you have to find five examples of when you did give somebody an 85% discount.
So that's cool if you have discrete values, but what if you have continuous numbers? Then you may have to group them up, this is called discretization. And then see, if when discrete bands you have at least five examples of each band.

Take the second one. The date that a promotional offer starts. Assuming that you may have to group things, all promotional offers that start in January.
Do you have at least five promotional offers that you started in January? Do you have at least five promotional offers that you started in February? If you don't, you may have to group things again.
You may not be able to use date. You may not be able to use month. You may have to use quarter. Do you have at least five examples of things that started in Q one and Q two and queue three and Q four, for example?
You may have to group up your values so that you have enough examples of each value.

Number of customers who opened advertising email. Hopefully you have enough examples of that, irrespective of the number you picked.
You have different types of advertising emails, and you have some that were opened by 1,000 people, and some that were open by 1,200 people, and some that were opened by 8,000 people.
Maybe you will have enough until you get to the very tail end of your distribution, and then you have only one email that actually got opened by 15 million customers. And you know that's a outlier, you can't use 15 million in your dataset.

Predict whether a credit card transaction is fraudulent

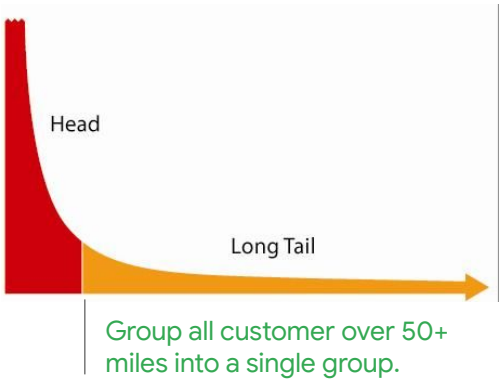
1. Whether the cardholder has purchased these items at this store before.
2. Distance between cardholder address and store.

Whether a card holder has purchased these items at this store before. This is to predict whether a credit card transaction is fraudulent.
Will you have enough examples of card holders who've purchased, and card holders who haven't purchased? It doesn't matter which item or which store because the way you're defining this, you should have enough customers who've purchased the item and enough customers who haven't.
Suppose you define this as whether the card holder, or customer has purchased a bag of diapers between 8 PM and 9 PM at this specific store. That is way too specific. So it really depends on how you define it. But if you define it general enough, such that you'd have enough examples of a value, then you're good.

Distance between a card holder address and the store. Will you have enough examples of customers who live ten miles away? You probably will. What about customers living 50 miles away? If maybe, then this starts becoming a problem.

Predict whether a credit card transaction is fraudulent

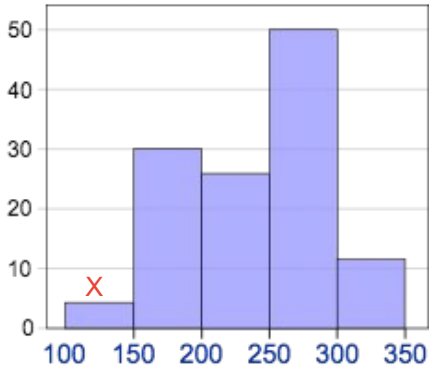
1. Whether the cardholder has purchased these items at this store before.
2. Distance between cardholder address and store.



This is basically where you may have to start grouping them together.
You can't use a value as is. You say, I'm going to take all the customers who live more than 50 miles away and I'll treat them as one lump. So you're not going to actually take 3,243 miles and use that in your training dataset. Because now your neural network will happily go and train that any time that somebody comes in from 3,243 miles away, whatever that person does is fine.
Because that one time that this person came and used their card, they didn't commit a fraud. So that's basically what you want to avoid. So we're talking about the values of the features, not the values of the labels.
If you're going to use rainfall amount as a feature, you've got to make sure that you have enough days that it rained one centimeter, two centimeters, three centimeters, etc.
How do you make sure that you have enough examples of a particular value?

Predict whether a credit card transaction is fraudulent

1. Whether the cardholder has purchased these items at this store before.
2. Distance between cardholder address and store.



Plot histograms of your input features. As a rule of thumb, make sure you have at least five. So try to look for at least five.

Predict whether a credit card transaction is fraudulent

1. Whether the cardholder has purchased these items at this store before.
2. Distance between cardholder address and store.
3. Category of item being purchased.
4. Online or in-person purchase?

You want to have enough examples. Consider this the category of the item being purchased. Sure, you'll have enough examples of that for any category that you choose.
Online purchase, in-person purchase. Again, you'll have enough examples of those, so those shouldn't be a problem.

Bring human insight to problem

You need to have subject matter expertise and a curious mind to think of all of the ways a data field could be used as a feature.

Remember that feature engineering is not done in a vacuum. After you train your first model you can always come back and add or remove features for model two.

Bring your human insight into the problem.

You need to have subject matter expertise and a curious mind to think of all of the ways a data field could be used as a feature. Remember that feature engineering is not done in a vacuum. After you train your first model you can always come back and add or remove features for model two.

Agenda

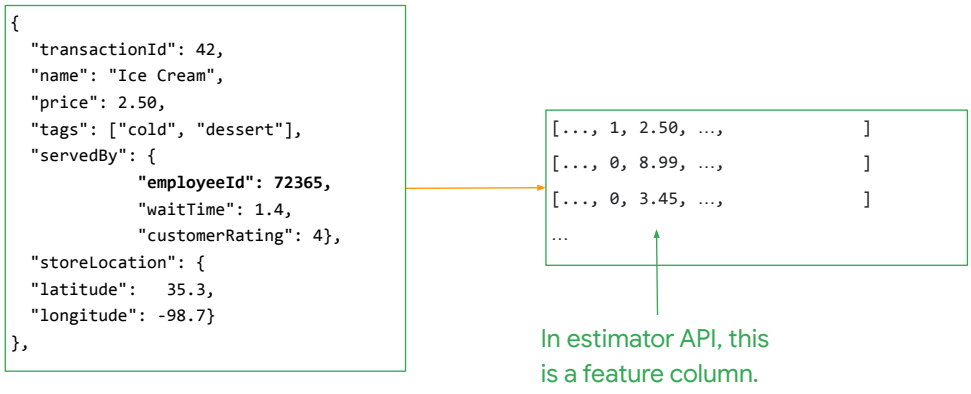
Turn raw data to features

Compare good versus bad features

Represent features

Next you're going to look at representing features with some examples.

Raw data are converted to numeric features in different ways



In this example, this is your raw data. You're in an ice cream store. You're trying to figure out if your ice cream is served by some employee and if the customer waited 1.4 seconds, or 1.4 minutes.
You want want to predict the rating. How satisfied is your customer going to be based on who served them, how long they waited, what it is that they bought, what the store location was, etc.?
This is your training data. You would take this training data and make them all numbers.

Because neural networks deal with numbers. You'll take your data, make them all numbers and those are your features. So in TensorFlow you're going to take this thing, which is a JSON input, comes out of your web application ultimately goes into a data warehouse, you pull it out, you create numeric values.
In TensorFlow each of these columns is a feature column.
So, how do you take some data like this and make them feature columns? Make them numeric.

Numeric values can be used as-is

```
{
  "transactionId": 42,
  "name": "Ice Cream",
  "price": 2.50,
  "tags": ["cold", "dessert"],
  "servedBy": {
             "employeeId": 72365,
             "waitTime": 1.4,
             "customerRating": 4},
  "storeLocation": {
  "latitude":   35.3,
  "longitude": -98.7}
},
```



```
[... , 2.50, ..., 1.4 , ... ]
...
```

```
INPUT_COLUMNS = [
   ...,
   tf.feature_column.numeric_column('price'),
   ...
   ]
```

numeric_column is a type of feature column.

There are values like price or wait time. These are already numeric.
That's easy to encode.

You take them and use them as is. They're numeric, they have meaningful magnitude. So 2.5, 1.4 intense or flow in TF learn. This is what is called a real valued column. So you'll just say layers are real-valued-column 'price'. Layers that real-valued-column wait time. So these number that you used as is, they'll just be real-valued-columns.

Overly specific attributes should be discarded

```
{
  "transactionId": 42,
  "name": "Ice Cream",
  "price": 2.50,
  "tags": ["cold", "dessert"],
  "servedBy": {
             "employeeId": 72365,
             "waitTime": 1.4,
             "customerRating": 4},
  "storeLocation": {
  "latitude":   35.3,
  "longitude": -98.7}
},
```

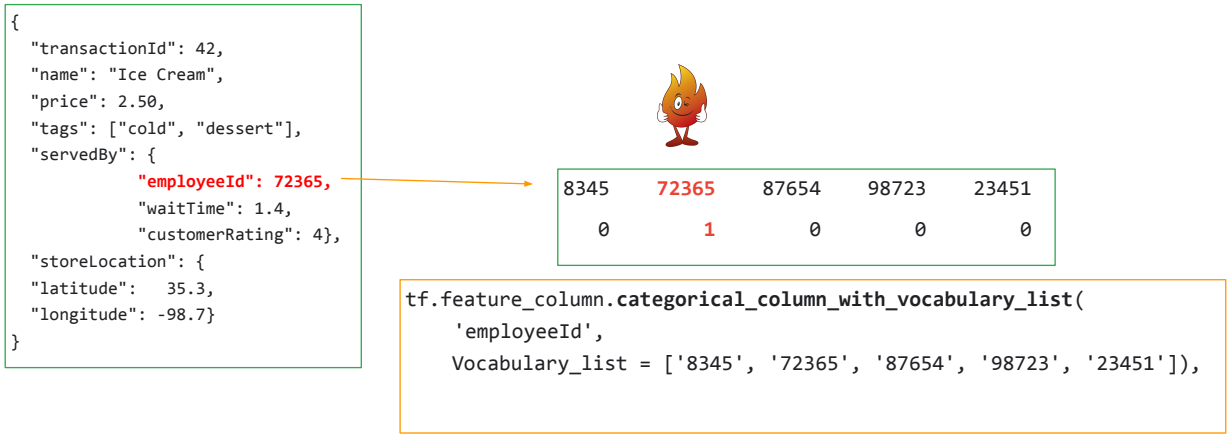
X

?

How about this input? Transaction ID is 42. No, that's way too specific. Throw it out, it can't be used as a feature.

Note: categorical_column_with_vocabulary_list is a type of feature column
How about employee ID? Employee ID is 72365. Is that numeric?
It's a number, but does it have meaningful magnitude? Is somebody with an employee ID of 72365 twice as good as somebody with an employee ID of 36182. No. Right?
So you can't use the employee ID as it is. You have to do something with it.

Categorical variables should be one-hot encoded



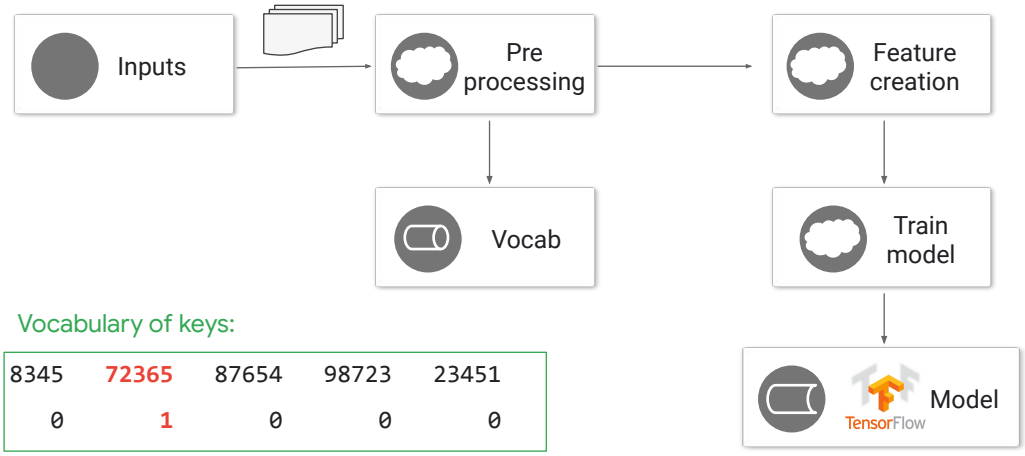
Suppose your ice cream shop has five employees, employee number 8345, employee number 72365 etc. What you can do say, if this is employee number 72365, I'll represent this employee ID by this vector.

The vector 01000 because I've defined the second column as corresponding to employee 72365. So essentially, I make it like a bit mask. You make that employees column one and all other columns zero.

This is called One-hot encoding. There's one column that's hot and all the other columns are cold. If you have five employees in an ice cream store, you essentially have five columns. In TensorFlow, this is called a sparse column. You basically say that I want to create a sparse column with the keys. The column name is employee ID and the keys are 8345, 72365 etc. You just pass the strings for each of those keys.

TensorFlow will take the string that you provide a direct training time, our prediction time and represent them. One-hot encode them. Make them all numeric. What you do is say that employee ID is a sparse column. This is if you know the keys beforehand.

Preprocess data to create a vocabulary of keys



What if you don't know the keys beforehand? What you would have to do is that you would take your input data, you would preprocess this data to find out all the keys that occur in your training dataset, and create a vocabulary of keys.

So that's your first step, that's the preprocessing.

You'd have to do this before you do your training. And then, you would create a new dataset where these preprocessed values can then be used. So, before you ever get to training the model, you need to create a vocabulary, and this vocabulary needs to be available at prediction time because at prediction time, the user is going to come back and say employee ID 72365, and the model needs to know that at training time, it decided that 72365 was the second column.

The vocabulary and the mapping of the vocabulary needs to be identical at prediction time

8345	72365	87654	98723	??????
0	0	0	0	0

The vocabulary needs to be identical and the mapping of the vocabulary needs to be identical at prediction time. Otherwise, it's not going to work.

So consider what would happen if you hire a new employee?

Would the model still be the same?
At this point, you don't have a place to put this new employee. So what this means is that you're unable to predict for this new employee. This is the kind of thing that you need to think about beforehand, and you might add something around what do I do with an employee that I don't know about – an employee that isn't found – and then you decide that perhaps you're going to find the average of all your current employees and use them.

Meanwhile, you collect data about the times that this employee is on duty and the customer satisfaction associated with this employee with different wait times and different things that they're serving. Once you've collected that, you use that in your prediction.

Options for encoding categorical data

If you know the keys beforehand:

```
tf.feature_column.categorical_column_with_vocabulary_list('employeeId',
    vocabulary_list = ['8345', '72345', '87654', '98723', '23451']),
```

If your data is already indexed; i.e., has integers in [0-N):

```
tf.feature_column.categorical_column_with_identity('employeeId',
    num_buckets = 5)
```

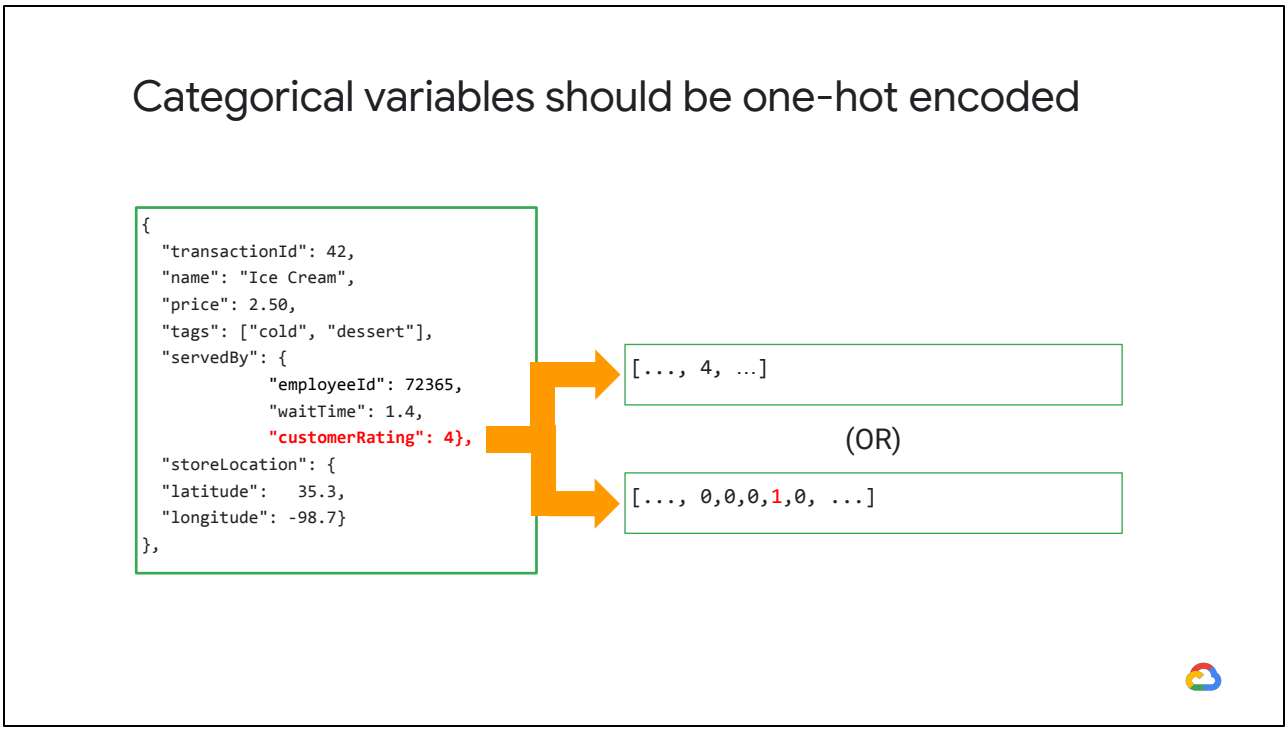
If you don't have a vocabulary of all possible values:

```
tf.feature_column.categorical_column_with_hash_bucket('employeeId',
    hash_bucket_size = 500)
```

So, if you know the keys beforehand, you essentially create sparse column with keys and you pass in the keys, you just hard code the keys.
These are all different ways of creating a sparse column.

Sometimes your data might already be indexed. Maybe for example you have Employee ID and they just happen to be numbers one to 1000. At that point they're already indexed. They're not arbitrarily big numbers all over the place, they're just one to n. If that's the case, you say I want to create a sparse column with the integerized feature which is Employee ID, and there are five employees.
So where this is useful in your taxi example is that you'll use it for the hour of the day, because that's automatically integerized from 0 to 23.
It's perfect as an integerized feature, because the hour of the day is not completely numeric because the number like 23 is very close to the number 0 1. It's only 2 hours away.

Take the third possibility. Suppose you don't have a vocabulary and it's not integerized.
You don't want to go out, build a vocabulary, and you don't really care. So what you do is that you say, I'm going to take my employee ID, hash it, compute the hash of the employee ID and just break that hash up into 500 buckets. Why would you do this? Suppose your company has 100 employees and you hash it to 500 buckets. So on average, each bucket will either have zero employee or one employee in it. It's almost like a one-hot encoding, but a 500-hot encoding, that kind of gets the same thing, but without having to build a vocabulary.



Thinking back to the customer rating example. What do you do with it? If you are trying to predict the customer rating then it's a label and you're not even worried. But say that you're trying to use this as an input because you're trying to predict something else. So if you have something like a rating and you want to use it as an input feature you could do one of two things.

You could treat it like a continuous number, 1-5. It's numeric. It sort of has a meaningful magnitude for us more than three or you can say that four stars is very different from five stars is very different from two stars in which case I am going to one-hot encode it. So, in some cases you have choices, to the customer rating; you can one-hot encode it or you can treat it as a number. Up to you, choose how to deal with the rating.

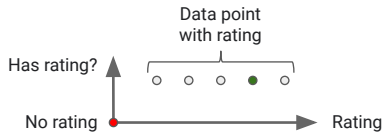
Don’t mix magic numbers with data

```
{
  "transactionId": 42,
  "name": "Ice Cream",
  "price": 2.50,
  "tags": ["cold", "dessert"],
  "servedBy": {
              "employeeId": 72365,
              "waitTime": 1.4,
              "customerRating": -1},
  "storeLocation": {
  "latitude":   35.3,
  "longitude": -98.7}
},
```

```
[..., 4,1, ...]  # 4
[..., 0,0, ...]  # -1
```

(OR)

```
[..., 0,0,0,1,1, ...]  # 4
[..., 0,0,0,0,0, ...]  # -1
```



One thing to watch out for is what you do if the customer didn't actually provide a rating? That you might be doing a survey and the customer didn't answer that survey? What do you do with missing data?

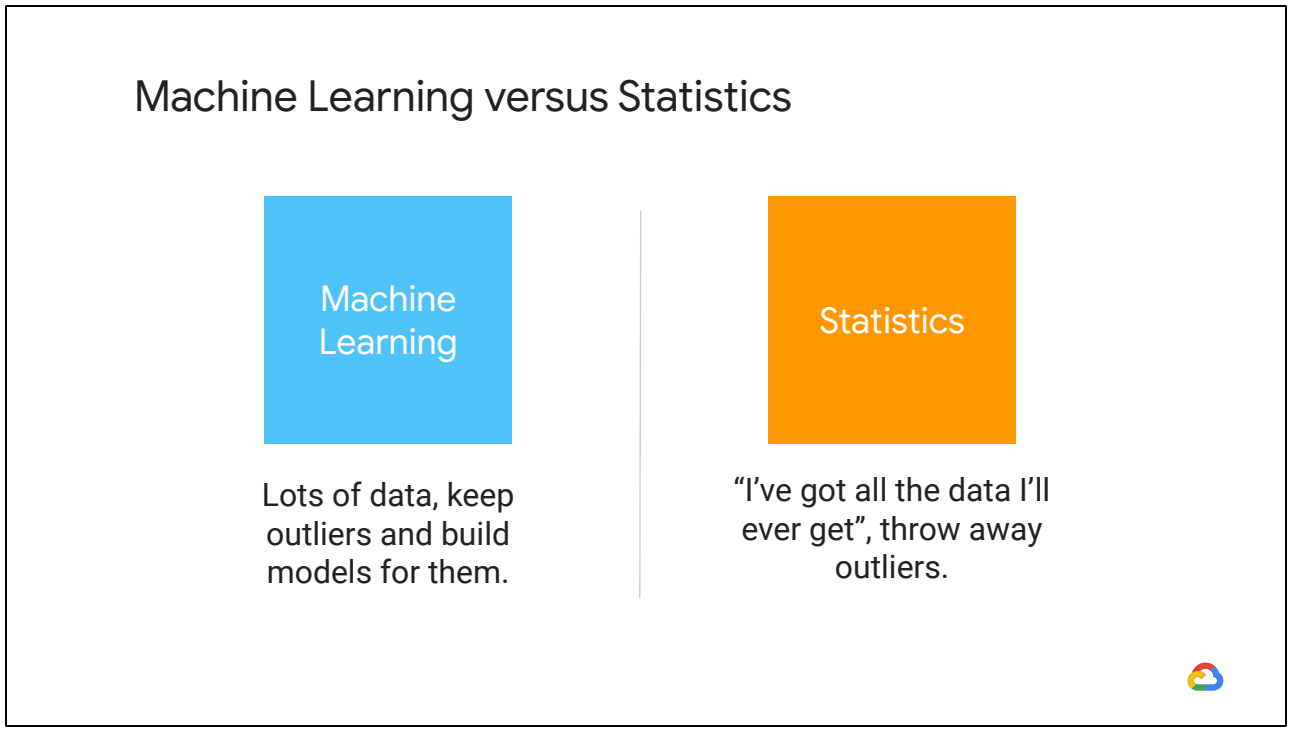
One option is to use two columns. One for the rating, and one for whether or not you got a rating.
In this case, the number 4 is a rating that a customer gave you. And 1 means that they gave you a rating. A 0 is they didn’t rate you.

You could also do it in the other way if you’re doing one-hot encoding.
You would say I got a rating of 4 (0 0 0 1) or I did’nt get a rating (0 0 0 0). But don’t make the mistake of not having the second column. You don’t want to mix magic numbers with real values. You have to add an extra column to state whether or not you observed the value or not.
If you have missing data you need to have another column.

Agenda

Feature Engineering

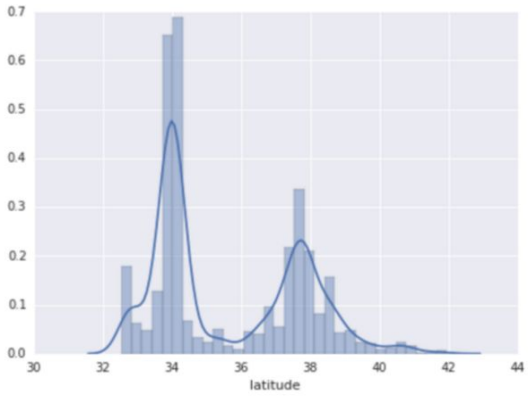
Next you're going to look at representing features with some examples.



If you’ve taken statistics, you might say for those missing values you would normally impute a value like the average for that column. This is where philosophically ML and statistics start to diverge.
In ML, the idea is that you build a separate model for the situation where you have data vs when you don't. We can afford to do this in ML on where we have data and where we don't have data because in ML we have enough data we want to build something as fine-grain where we can.

Statistics on the other hand is about keeping the data that you have and getting the best results out of the data you have. This difference in philosophy affects how you treat outliers. In ML you go out and find enough outliers so that it becomes something you train with. With statistics you say “I’ve got all the data I’ll be able to get” so you throw out outliers. It’s a philosophical difference because of the scenarios where ML and statistics are used. Statistics is often used in a limited data regime and ML operates with lots of data so having an extra column to flag where you're missing data is what you do. When you don’t have enough data you impute or replace by average.

Exact floats are not meaningful



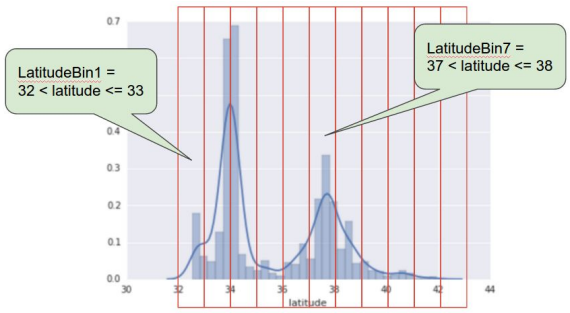
LOS ANGELES

SAN FRANCISCO

The example shown here is of predicting house value.

The dataset includes latitude. Two peaks -- one for SFO and the other for LAX. It doesn't make sense to represent latitude as a floating-point feature in our model. That's because no linear relationship exists between latitude and housing values. For example, houses in latitude 35 are not 35/34 more expensive (or less expensive) than houses at latitude 34. And yet, individual latitudes probably are a pretty good predictor of house values.

Discretize floating point values into bins

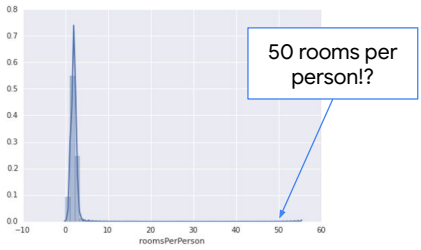



```
lat = tf.feature_column.numeric_column('latitude')
dlat = tf.feature_column.bucketized_column(
         lat, boundaries=np.arange(32,42,1).tolist()
```

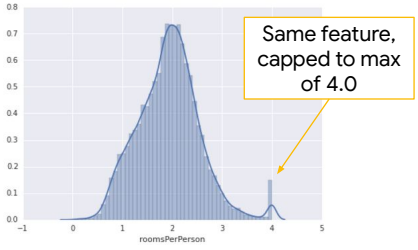
Now consider this. Instead of having one floating-point feature, you now have 11 distinct boolean features (LatitudeBin1, LatitudeBin2, LatitudeBin11).

Here, you use fixed bin boundaries. Another option is to use quantile boundaries so that the number of values in each bin is constant.

Crazy outliers will hurt trainability



Rooms Per Person



Capped Rooms Per Person

```
features['capped_rooms'] = tf.clip_by_value(
    features['rooms'] ,
    clip_value_min=0,
    clip_value_max=4
)
```

Especially in regression problems, quite a few training cycles will be spent trying to get the unusual instances correct.

Collapse the long tail (in ML) vs remove from the set (normal statistics). E.g. if the house has 50 rooms you would set it to have 4 rooms (our top in the range).

Ideally, features should have a similar range

Typically `[0,1]` or `[-1,1]`

```
features['scaled_price'] =
      (features['price'] - min_price) / (max_price - min_price)
```

The idea is that the price of a home is in the 100's of thousands while number of rooms are small numbers. Optimizers have traditionally had a hard time dealing with this, the price ends up dominating the gradient.

Modern architectures end up taking variable magnitudes into account because of batch normalization, although you might run into issues if a batch of examples happens to all contain unusual values. So, this is not as important as it used to be.

BigQuery preprocessing - Feature Engineering

1 Representation transformation

Converting a numeric feature to a categorical feature (through bucketization) and converting categorical features to a numeric representation (through one-hot encoding, learning with counts, sparse feature embeddings, and so on). Some models work only with numeric or categorical features, while others can handle mixed type features. Even when models handle both types, they can benefit from different representation (numeric and categorical) of the same feature.

2 Feature construction

Creating new features either by using typical techniques, such as polynomial expansion (by using univariate mathematical functions), or feature crossing (to capture feature interactions). Features can also be constructed by using business logic from the domain of the ML use case.

BigQuery preprocessing involves two aspects:

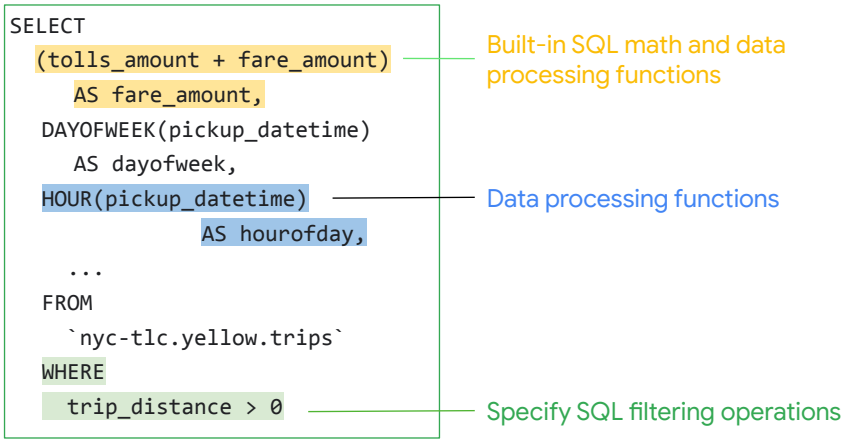
1. Representation transformation
2. Feature construction

Feature representation is converting a numeric feature to a categorical feature (through bucketization), and converting categorical features to a numeric representation (through one-hot encoding, learning with counts, sparse feature embeddings, and so on). Some models work only with numeric or categorical features, while others can handle mixed type features. Even when models handle both types, they can benefit from different representation (numeric and categorical) of the same feature.

Feature construction is creating new features either by using typical techniques, such as polynomial expansion (by using univariate mathematical functions) or feature crossing (to capture feature interactions). Features can also be constructed by using business logic from the domain of the ML use case.

Basic feature engineering

Example of preprocessing in BigQuery



BigQuery can help with feature engineering because it lets you use SQL to implement common preprocessing tasks. For example, if you are preprocessing a dataset with records of taxi rides in New York City, you can specify SQL filtering operations to exclude bogus data from your training examples datasets, like the rides with a distance of 0 miles.

Built-in SQL math and data processing functions are also valuable for simple calculations additions over source data.

Data processing functions are also valuable for parsing common data formats, like timestamps, to extract details about the time of day.

Filter out bogus data; for example trip distances must be above zero. Extract hourly data. Perform calculations to get a new field: fare_amount.

Basic feature engineering
Example of dates and time

```
EXTRACT(DAYOFWEEK FROM  pickup_datetime) AS dayofweek,

EXTRACT(HOUR FROM pickup_datetime) AS hourofday,
```

```
CONCAT(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING), CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING)) AS hourofday,
```

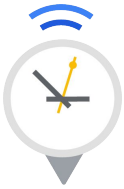
For example, here are three types of preprocessing for dates using SQL in BQML:

- Extracting the parts of the date into different columns: Year, month, day, etc.
- Extracting the time period between the current date and columns in terms of years, months, days, etc.
- Extracting some specific features from the date: Name of the weekday, weekend or not, holiday or not, etc.

Basic feature engineering
Example of dates and time

DATA STUDIO OUTPUT: From EXTRACT dates/time queries

	dayofweek	hourofday
1.	Week 6	4 AM
2.	Week 5	3 AM
3.	Week 4	2 AM
4.	Week 7	1 AM
5.	Week 3	12 AM



Note
For all non-numeric columns other than TIMESTAMP, BigQuery ML performs a one-hot encoding transformation that generates a separate feature for each unique value in the column.

Instructor Note: Unhide or hide based on audience level.

Here is an example of the dayofweek and hourofday queries extracted using SQL and visualized as a table in Data Studio. Note: For all non-numeric columns other than TIMESTAMP, BigQuery ML performs a one-hot encoding transformation. This transformation generates a separate feature for each unique value in the column.

BQML labs

Feature Engineering in BQML

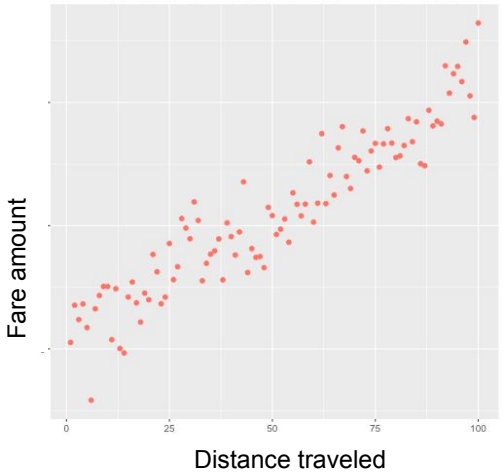
- Basic feature engineering
- Advanced feature engineering

So let's go ahead and learn how to incorporate feature engineering in a BigQuery machine learning model. We will use the New York taxi driver dataset to examine the effect of feature engineering on model prediction.

Regression problem: Predicting taxi fare

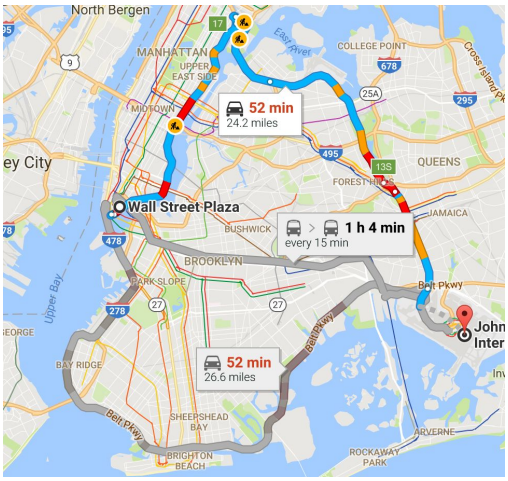
Problem: Predict taxi fare amount based on distance travelled

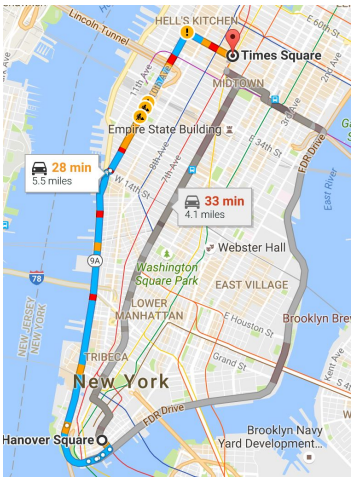
What is the error measure to optimize? RMSE

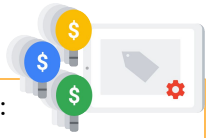


Our machine learning problem is a regression problem. The measure we will use is the root mean square error, or RMSE. There is additional content on RMSE in the labs.

ML problem: Estimating taxi fare







Taxi fare:

Fare amount
+
Distance travelled
+
Number of passengers
+
Pickup and dropoff location
+
Pickup date and time

http://www.nyc.gov/html/tlc/html/passenger/taxicab_rate.shtml

Our machine learning problem is to predict taxi fare price in New York city using BQML.

Baselines are important; It helps to know what error metric is “reasonable” and/or “good” for the problem

A baseline helps you set a goal for a good value for the error metric.

Often a simple heuristic rule can function as a good benchmark.

We will begin with a baseline model to help us set a goal for a good value for the error metric. Baselines are important to know because they help us determine a reasonable or good metric for the problem.

Evaluate model

```
SELECT * FROM ML.EVALUATE(MODEL feat_eng.baseline_model)
```

mean_absolute_error	mean_squared_error	mean_absolute_log_error	median_absolute_error	R squared
5.21	68.88	0.2581	3.790	.2261

(BQML splits the training data and reports evaluation statistics on the held-out set)

RMSE

```
SELECT SQRT(mean_squared_error) AS rmse FROM ML.EVALUATE(MODEL
feat_eng.baseline_model)
```

rmse
08.299

The primary evaluation metric for this ML problem is the root mean-squared error. RMSE measures the difference between the predictions of a model and the observed values.

Here is the code you will use to evaluate the model.

Note that BigQuery automatically split the data we gave it and trained on only a part of the data, using the rest for evaluation. After creating your model, you evaluate the performance of the regressor using the ML.EVALUATE function. The ML.EVALUATE function evaluates the predicted values against the actual data.

Root mean squared error (RMSE): The primary evaluation metric for this ML problem is the root mean-squared error. RMSE measures the difference between the predictions of a model and the observed values. A large RMSE is equivalent to a large average error, so smaller values of RMSE are better. One nice property of RMSE is that the error is given in the units being measured, so you can tell very directly how incorrect the model might be on unseen data.

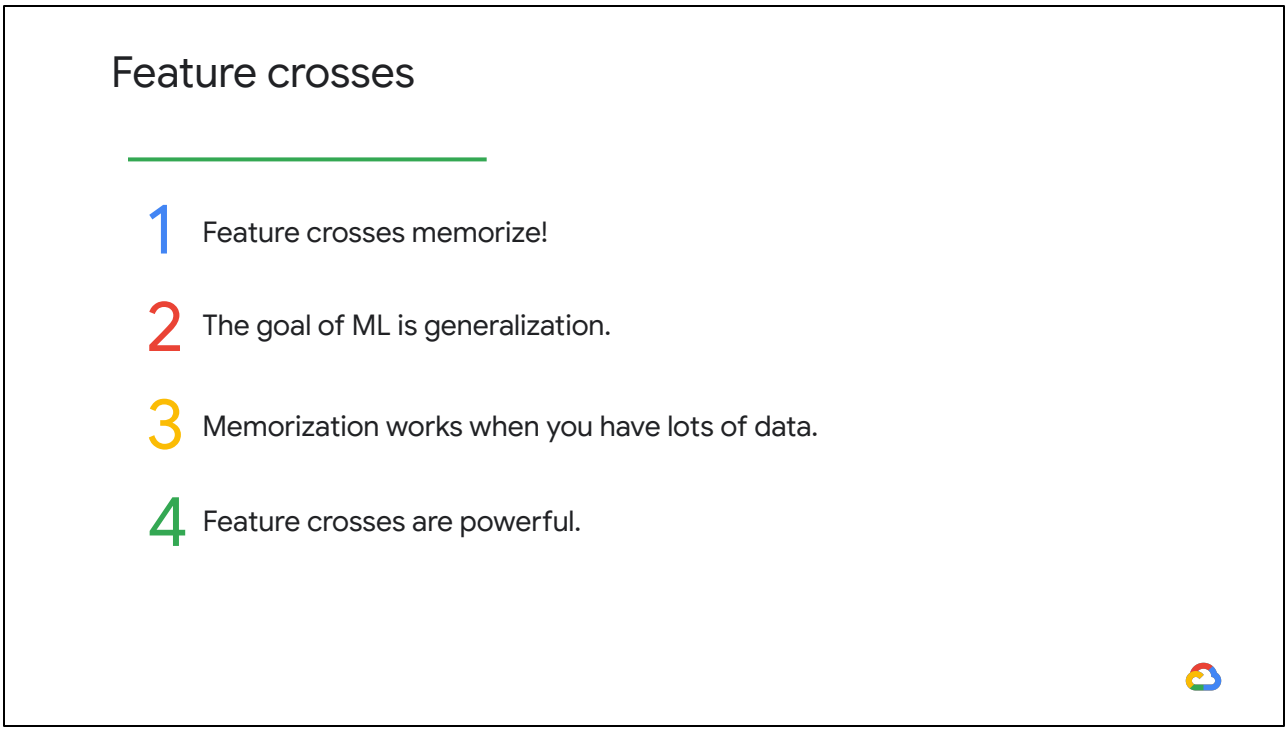
Advanced feature engineering

BQML preprocessing functions

- ML.FEATURE_CROSS(STRUCT(features)) does a feature cross of all the combinations.
- ML.POLYNOMIAL_EXPAND(STRUCT(features), degree) creates x, x2, x3, etc.
- ML.BUCKETIZE(f, split_points) where split_points is an array.

Here are some of the preprocessing functions in BigQuery ML:

- ML.FEATURE_CROSS(STRUCT(features)) does a feature cross of all the combinations.
- ML.POLYNOMIAL_EXPAND(STRUCT(features), degree) creates x, x2, x3, etc.
- ML.BUCKETIZE(f, split_points) where split_points is an array.



Feature crosses are about memorization. Memorization is the opposite of generalization, which is what machine learning aims to do.

So, should you do this?

In a real-world ML system, there is place for both.

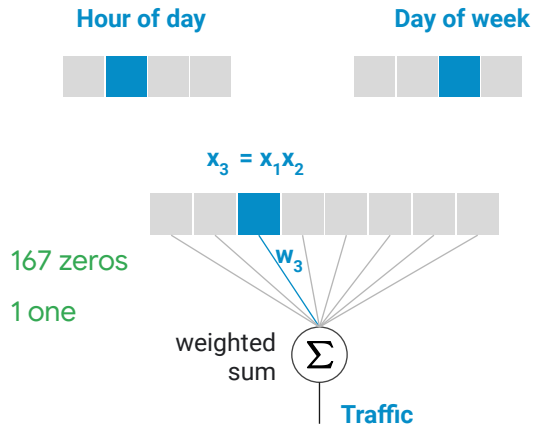
Memorization works when you have so much data that, for any single grid cell within your input space, the distribution of data is statistically significant. When that is the case, you can memorize. You are essentially “just” learning the mean for every grid cell.

Deep learning also needs a lot of data; whether you want to feature cross or you want to use many layers, you need a lot of data.

If you are familiar with traditional machine learning, you may not have heard much about feature crosses because they memorize and only work on large datasets.

You will find feature crosses extremely useful in real-world datasets. Larger data allows you to make your boxes smaller, and you can memorize more finely. Feature crosses are a powerful feature preprocessing technique on large datasets.

Feature crosses lead to sparsity



Allow the model to learn traffic patterns by creating a new feature that combines the time of day and day of week (this is called a feature cross).

Our ML lab model would be greatly improved if, instead of treating the hour-of-day and day-of-week as independent inputs, we essentially concatenated them to create a feature cross. Here is an example. For any particular row of your input dataset, how many nodes in x3 are “lit up”? Just one. Do you see why?

Every label, every observation of table, is taken at a specific time. That corresponds to a specific hour on a specific day of the week. So, 3pm in the hour-of-day input and Wednesday in the day-of-week input.

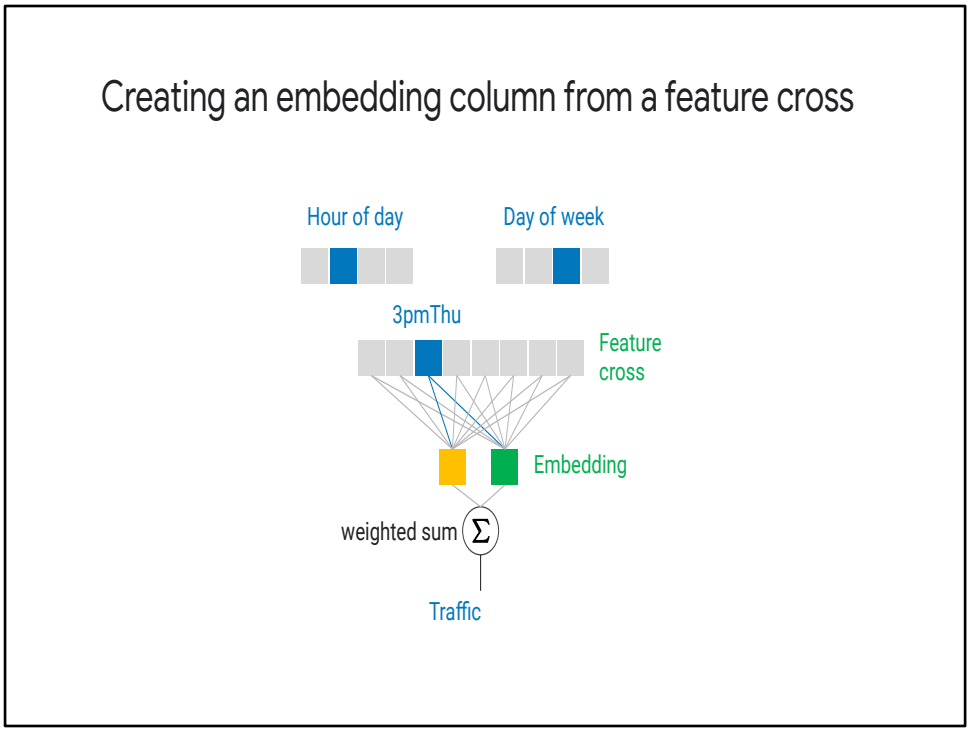
Feature cross these, and what do you have? You have one input node: the input node that corresponds to 3pm on Wednesday will be 1. All the other input nodes for x3 will be zero.

The input, therefore, will consist of 167 zeros and one 1. **That is the definition of sparsity: a feature with mostly missing values, in our case “zerio.”**

When you do a feature cross, the input is very, very sparse. TensorFlow will give us easy tools to deal with this.
Note some observations about sparsity:

1. Sparse models contain fewer features and therefore are easier to train

1. on limited data.
2. Fewer features also means less chance of over fitting.
3. Fewer features also means it is easier to explain to users because only the most meaningful features remain.

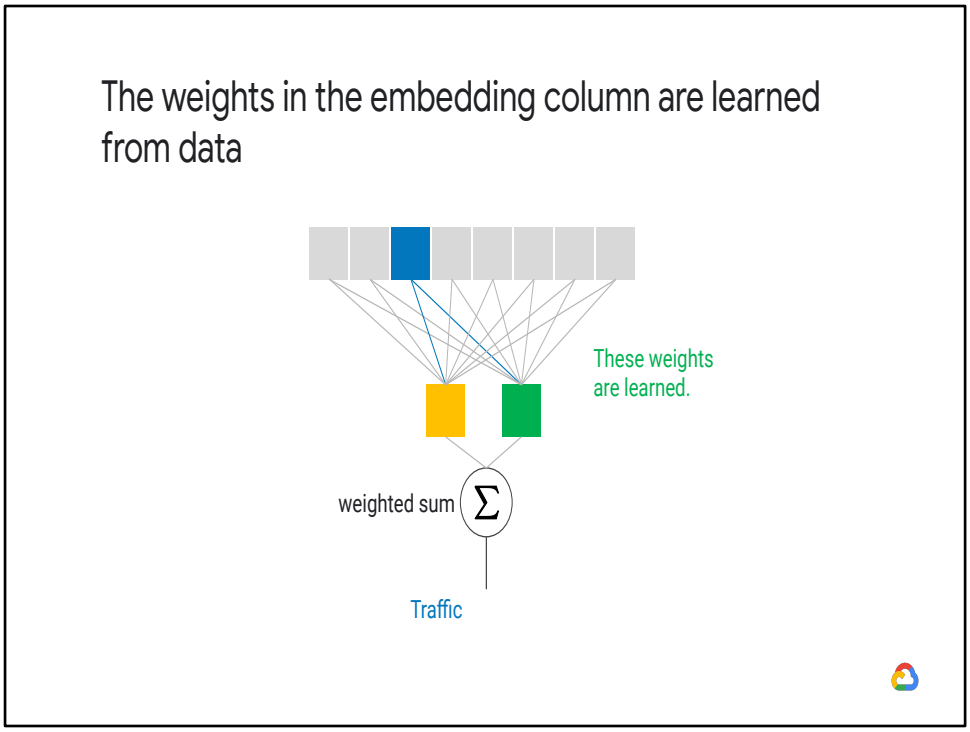


We can also create an embedding from a feature cross.

If we use a large number of hash buckets when doing this feature cross, we could be relatively confident that each of the buckets contained only one hour-day combination. So, instead of one-hot encoding the feature cross and using it as-is, we could pass it through a dense layer and then train the model to predict traffic as before.

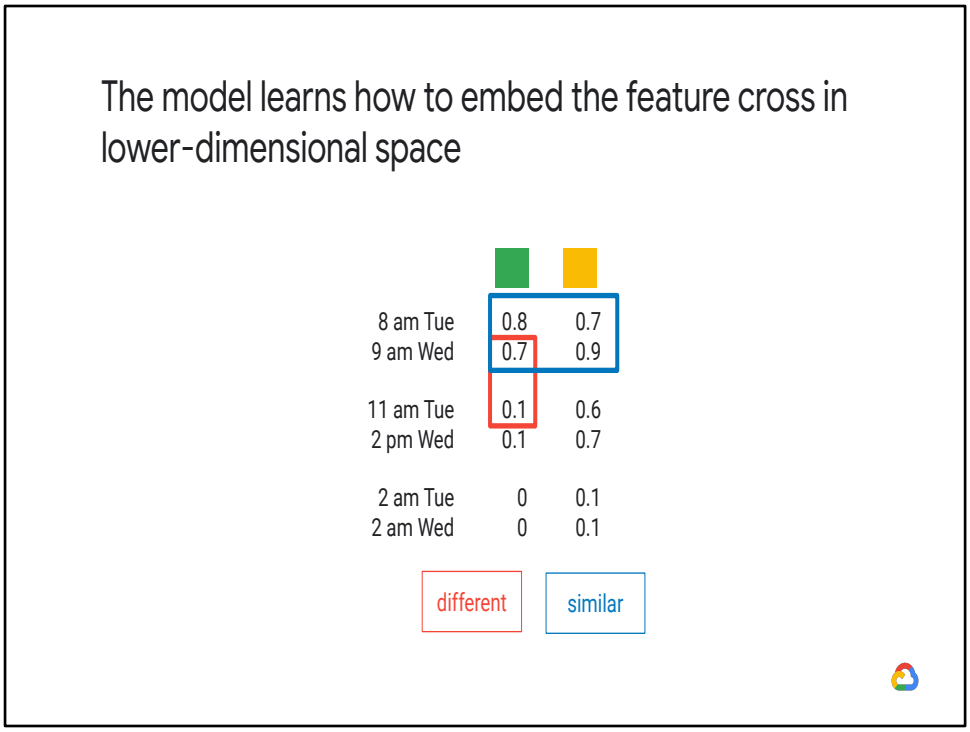
This dense layer, shown by the yellow and green nodes, here creates an embedding.

The embeddings are real-valued numbers because they are a weighted sum of the feature crossed values.



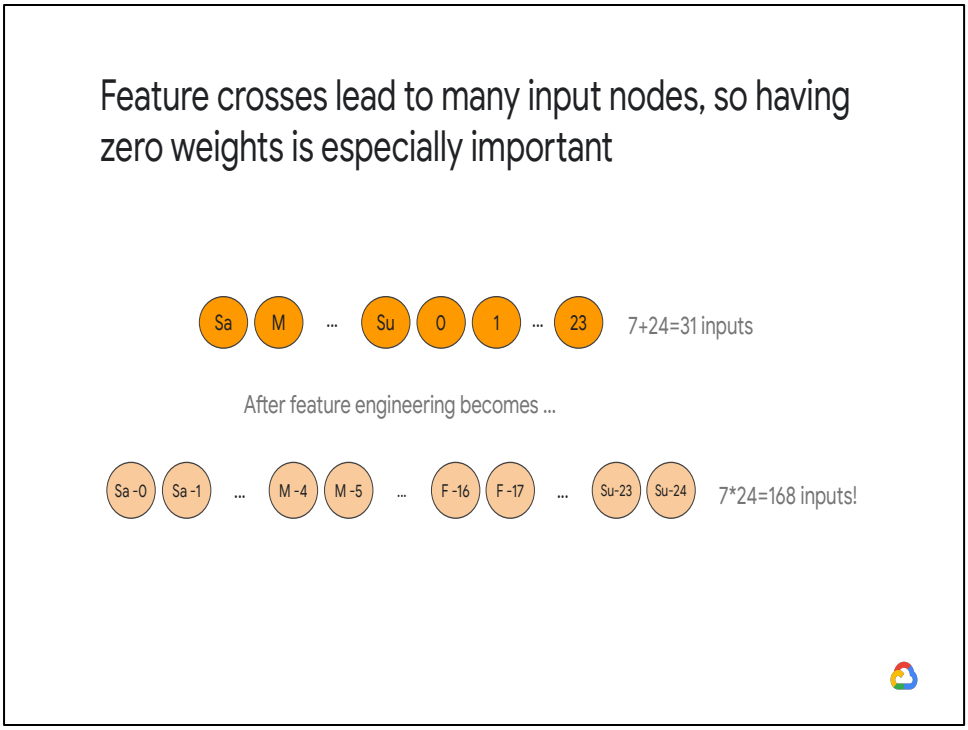
The weights that go into the embedding layer (the yellow and green nodes) are learned from the data.

The point is that by training these weights on a dataset, to solve a useful problem, something neat happens …
The feature cross of day-hour has 168 unique values, but we are forcing it to be represented with just two real-valued numbers. So ...



the model learns how to embed the feature cross in lower-dimensional space. It's important that all the information in the hour-of-day and day-of-week as it pertains to traffic at city intersections is shoehorned into just two numbers.

If you do this on a large enough and good enough dataset, these numbers have one very useful property:
Times that are similar in terms of traffic get real-valued numbers that are close together.
And times are different in terms of traffic get real-valued numbers that are different.



Many machine learning models already have enough features. For instance, let's say that I have data that contains the datetime of orders being placed. A first order model would probably include 7 features for the days of the week and 24 features for hours of the day, plus possibly many other features. Therefore, the day of the week plus hour of the day is already 31 inputs.

Now what if we want to look at the second order effects of day of the week crossed with hour of the day? That is another 168 inputs in addition to our 31 plus others, for a grand total now of almost 200 features just for that one datetime field plus whatever other features we are using.

If we cross this with the one hot encoding for U.S. state, for example, the triple cartesian product is already at 8,400 features, with many of them probably being very sparse of mostly zeroes. Hopefully this makes clear why built-in feature selection through L1 regularization can be a very good thing.

What strategies in addition to L1 regularization can we use to remove feature coefficients that aren't useful?

We could include using simple counts of which features occur with non-zero values.

Linear for sparse, independent features

```
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]
```

This is what a sparse matrix looks like: very, very wide, with lots and lots of features. You want to use linear models to minimize the number of free parameters. And if the columns are independent, linear models may suffice.

Spatial and temporal features

Spatial features

```
ST_Distance(ST_GeogPoint(pickuplon, pickuplat), ST_GeogPoint(dropofflon, dropofflat))
AS euclidean
```

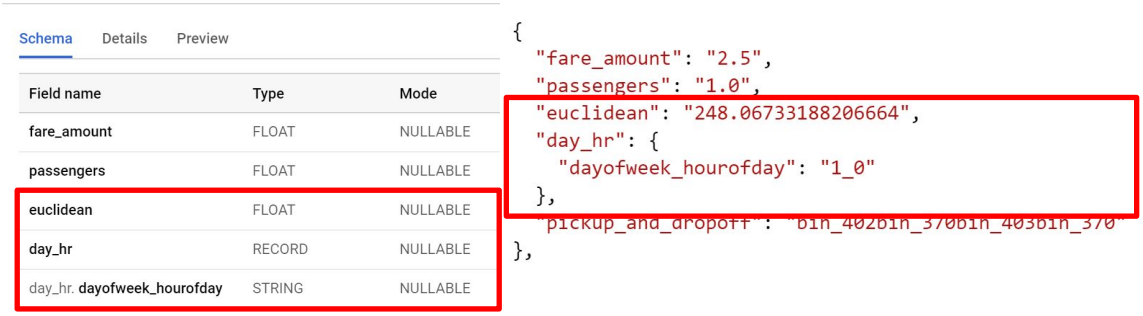
Temporal features

```
ML.FEATURE_CROSS(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING),
CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
```

Look at how ML_FEATURE_CROSS is used to extract temporal features.

Feature cross

```
Spatial and Features
```



Schema | Details | Preview

Field name	Type	Mode
fare_amount	FLOAT	NULLABLE
passengers	FLOAT	NULLABLE
euclidean	FLOAT	NULLABLE
day_hr	RECORD	NULLABLE
day_hr. dayofweek_hourofday	STRING	NULLABLE

If you go to the BigQuery console, you would see the SQL statements from the previous slide executed here. On the left is the schema showing the new features, and on the right is the JSON file showing the new features.

Note: BQML by default assumes that numbers are numeric features and strings are categorical features. We need to convert both the dayofweek and hourofday features to strings because the model (Neural Network) will automatically treat any integer as a numerical value rather than a categorical value. Thus, if not cast as a string, the dayofweek feature will be interpreted as numeric values (e.g. 1,2,3,4,5,6,7) and hourofday will also be interpreted as numeric values (e.g. the day begins at midnight, 00:00, and the last minute of the day begins at 23:59 and ends at 24:00). As such, there is no way to distinguish the "feature cross" of hourofday and dayofweek "numerically." Casting the dayofweek and hourofday as strings ensures that each element will be treated like a label and will have its own associated coefficient.

BUCKETIZE

Spatial Features

```
CONCAT(
    ML.BUCKETIZE(pickuplon, GENERATE_ARRAY(-78, -70, 0.01)),
    ML.BUCKETIZE(pickuplat, GENERATE_ARRAY(37, 45, 0.01)),
    ML.BUCKETIZE(dropofflon, GENERATE_ARRAY(-78, -70, 0.01)),
    ML.BUCKETIZE(dropofflat, GENERATE_ARRAY(37, 45, 0.01))
) AS pickup_and_dropoff
```

```
{
  "fare_amount": "2.5",
  "passengers": "1.0",
  "euclidean": "248.06733188206664",
  "day_hr": {
    "dayofweek_hourofday": "1_0"
  },
  "pickup_and_dropoff": "bin_402bin_370bin_403bin_370"
},
```

pickup_and_dropoff
bin_406bin_379bin_406bin_379
bin_402bin_376bin_402bin_376
bin_422bin_366bin_423bin_366
bin_404bin_378bin_405bin_380
bin_403bin_374bin_403bin_374
bin_401bin_376bin_406bin_368
bin_407bin_376bin_407bin_376
bin_404bin_378bin_404bin_377
bin_402bin_376bin_403bin_376
bin_402bin_376bin_402bin_376
bin_401bin_372bin_401bin_372
bin_401bin_375bin_401bin_375
bin_404bin_380bin_404bin_380

Bucketize is a preprocessing function that creates "buckets" (bins); that is, it bucketizes a continuous numerical feature into a string feature with bucket names as the value.

- ML.BUCKETIZE(feature, split_points)
- feature: A numerical column.
- split_points: Array of numerical points to split the continuous values in a feature into buckets. With n split points (s1, s2 … sn), n+1 buckets are generated.
- Output: The function outputs a STRING for each row, which is the bucket name. bucket*name is in the format of bin*
- Currently, our model uses the ST_GeogPoint function to derive the pickup and dropoff feature. In this lab, we use the BUCKETIZE function to create the pickup and dropoff feature.

TRANSFORM CLAUSE: BQML

```
CREATE OR REPLACE MODEL feat_eng.final_model
TRANSFORM(
  fare_amount,
    #SQRT( (pickuplon-dropofflon)*(pickuplon-dropofflon) + (pickuplat-dropofflat)*(pickuplat-dropofflat) ) AS euclidean,
      ST_Distance(ST_GeogPoint(pickuplon, pickuplat), ST_GeogPoint(dropofflon, dropofflat)) AS euclidean,
    ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING) AS dayofweek,
        CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
   CONCAT(
     ML.BUCKETIZE(pickuplon, GENERATE_ARRAY(-78, -70, 0.01)),
     ML.BUCKETIZE(pickuplat, GENERATE_ARRAY(37, 45, 0.01)),
     ML.BUCKETIZE(dropofflon, GENERATE_ARRAY(-78, -70, 0.01)),
     ML.BUCKETIZE(dropofflat, GENERATE_ARRAY(37, 45, 0.01))
   ) AS pickup_and_dropoff
)
OPTIONS(input_label_cols=['fare_amount'], model_type='linear_reg', l2_reg=0.1)
AS

SELECT * FROM feat_eng.feateng_training_data
```

Before we perform our prediction, we should encapsulate the entire feature set in a TRANSFORM clause. BigQuery ML now supports defining data transformations during model creation, which will be automatically applied during prediction and evaluation. This is done through the TRANSFORM clause in the existing CREATE MODEL statement. When the TRANSFORM clause is used, user-specified transforms during training will be automatically applied during model serving (prediction, evaluation, etc.)

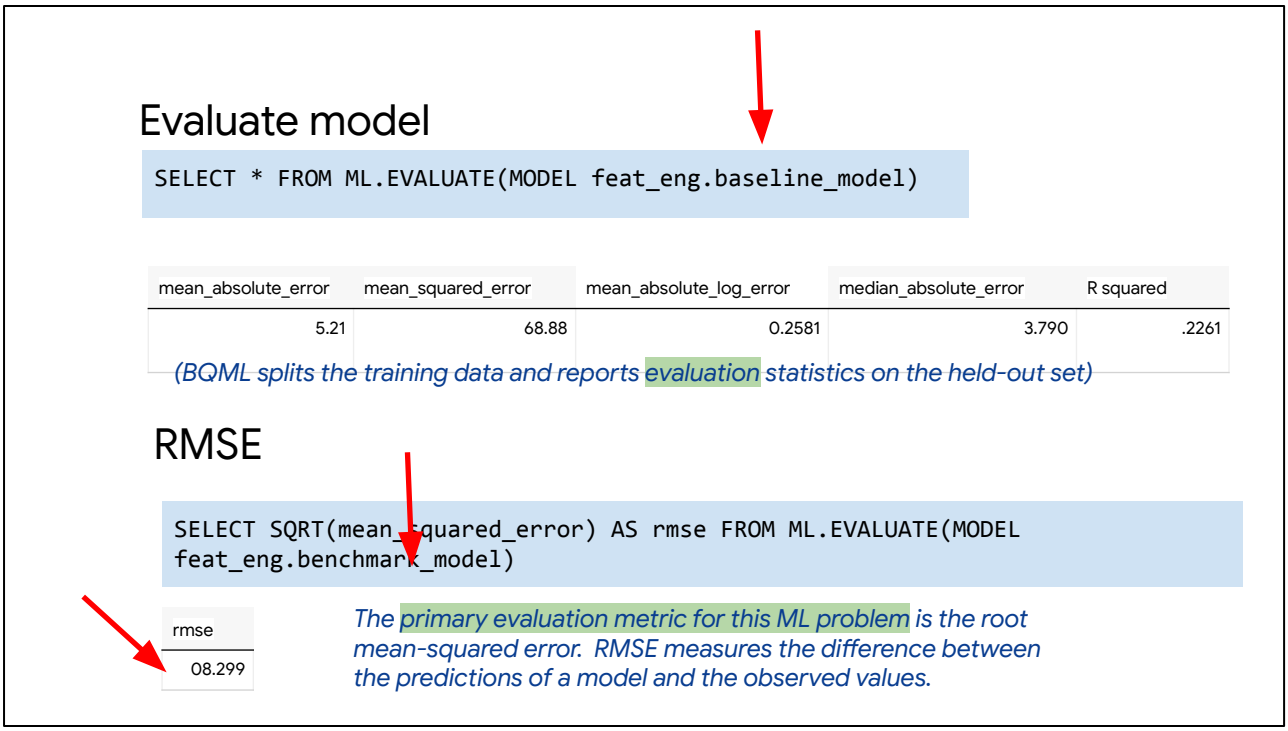
In our case, we are using the TRANSFORM clause to separate the raw input data from the TRANSFORMED features. The input columns of the TRANSFORM clause are the query_expr (AS SELECT part). The output columns of TRANSFORM from select_list are used in training. These transformed columns are post-processed with standardization for numerics and one-hot encoding for categorical variables by default.

TRANSFORM ensures that transformations are automatically applied during ML.PREDICT

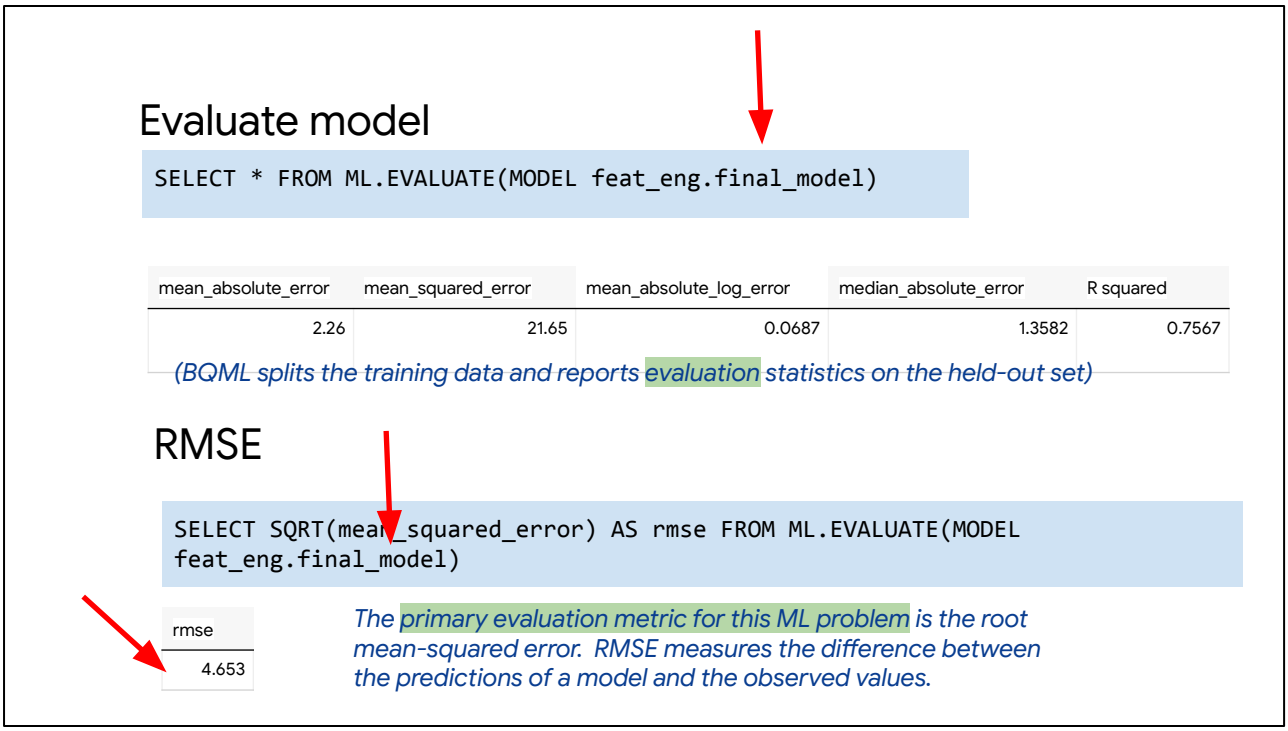
```
CREATE OR REPLACE MODEL feat_eng.final_model
TRANSFORM(
  fare_amount,
    #SQRT( (pickuplon-dropofflon)*(pickuplon-dropofflon) + (pickuplat-dropofflat)*(pickuplat-dropofflat) ) AS euclidean,
      ST_Distance(ST_GeogPoint(pickuplon, pickuplat), ST_GeogPoint(dropofflon, dropofflat)) AS euclidean,
    ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING) AS dayofweek,
        CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
   CONCAT(
     ML.BUCKETIZE(pickuplon, GENERATE_ARRAY(-78, -70, 0.01)),
     ML.BUCKETIZE(pickuplat, GENERATE_ARRAY(37, 45, 0.01)),
     ML.BUCKETIZE(dropofflon, GENERATE_ARRAY(-78, -70, 0.01)),
     ML.BUCKETIZE(dropofflat, GENERATE_ARRAY(37, 45, 0.01))
  ) AS pickup_and_dropoff
)
OPTIONS(input_label_cols=['fare_amount'], model_type='linear_reg', l2_reg=0.1)
AS

SELECT * FROM feat_eng.feateng_training_data
```

The advantage of encapsulating features in the TRANSFORM clause is that the client code doing the PREDICT doesn't change; that is, our model improvement is transparent to client code. Note that the TRANSFORM clause MUST be placed after the CREATE statement.

In summary, the TRANSFORM clause ensures that transformations are automatically applied during prediction.



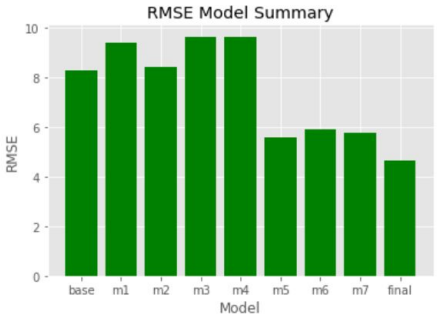
As you evaluate the model in the labs, notice that you begin with a baseline model for evaluation and your benchmark model to evaluate RMSE. The baseline model has no feature engineering.



The final model has feature engineering and will show a lower RMSE of $4.65 cents for a taxi fare.

RMSE: Summary table and visualization

Model	RMSE	Description
baseline_model	8.29	--Baseline model - no feature engineering
model_1	9.431	--EXTRACT DayOfWeek from the pickup_datetime feature
model_2	8.408	--EXTRACT hourofday from the pickup_datetime feature
model_3	8.328	--Feature cross dayofweek and hourofday -Feature Cross does lead ot overfitting
model_4	9.657	--Apply the ML.FEATURE_CROSS clause to categorical features
model_5	5.588	--Feature cross coordinate features to create a Euclidean feature
model_6	5.906	--Feature cross pick-up and drop-off locations features
model_7	5.75	--Apply the BUCKETIZE function
final_model	4.653	--Apply the TRANSFORM clause and L2 Regularization



Here is a table of all the models you will see in the lab.

Lab: Basic Feature Engineering in BQML

1. Create SQL statements to evaluate the model.
2. Extract temporal features.
3. Perform a feature cross on temporal features.

So let's go ahead and learn how to incorporate feature engineering in a BigQuery machine learning model. We will use the New York taxi driver dataset to examine the effect of feature engineering on model prediction.

The objectives of this lab are to:

1. Create SQL statements to evaluate the model
2. Extract temporal features
3. Perform a feature cross on temporal features

Steps:

1. Create the project dataset.
2. Create the feature engineering training table.
3. Create and evaluate the benchmark/baseline model.
4. Extract temporal features.
5. Perform a feature cross on temporal features.
6. Evaluate model performance.

Link to lab: [ML on GCP C4] Basic Feature Engineering in BQML (qwiklabs). github repo link here. CBL195

Lab: Advanced Feature Engineering BQML

1. Apply ML.FEATURE_CROSS to categorical features.
2. Create a Euclidian feature column.
3. Feature cross coordinate features.
4. Apply the BUCKETIZE function.
5. Apply the TRANSFORM clause and L2 Regularization.
6. Evaluate the model using ML.PREDICT.

The objectives of the second lab are to:

1. Perform a feature cross using BigQuery's ML.FEATURE_CROSS.
2. Derive a coordinate feature.
3. Feature cross coordinate features.
4. Code cleanup.
5. Apply the BUCKETIZE function.
6. Apply the TRANSFORM clause.
7. Apply L2 Regularization.
8. Evaluate model performance.
9. Create a predictive model.

Link to Lab: [ML on GCP C4] Performing Advanced Feature Engineering in BQML (qwiklabs). Github repo here.CBL196

BQML feature engineering summary

- Remove examples that you don't want to train on.
- Compute vocabularies for categorical columns.
- Compute aggregate statistics for numeric columns.
- Consider advanced feature engineering using ML.FEATURE_CROSS, TRANSFORM, and BUCKETIZE.

In BigQuery

This covers operations like excluding some data points from the training dataset.

Notice that to compute the re-scaling formula shown on the screen, you need to know the computing statistics and vocabularies over the entire input dataset.

Keep in mind that for some features, you will need statistics over a limited time window, like the number of products sold over the past hour. For these types of time-windowed features, you will use Beam's batch and streaming data pipelines.

Other features that can be preprocessed one data point at at time can be implemented either in TensorFlow directly or using Beam. Apache Beam, and the complementary Google Cloud technology called Dataflow, will be important to this part of the module.

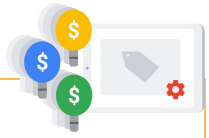
Keras labs

Feature engineering in Keras

- Basic feature engineering
- Advanced feature engineering

Next, we look at feature engineering in Keras.

ML problem: Estimating housing price



Housing Price?:

Median house
+
Longitude
+
Latitude
+
Number of rooms
+
Number of bedrooms

Our machine learning problem is to predict housing prices in California. One caveat: the data set is from the 1990 census, so prices have increased dramatically since then!

Wrap the dataframe with tf.data

```
def df_to_dataset(dataframe, shuffle=True, batch_size=32):
  dataframe = dataframe.copy()
  labels = dataframe.pop('median_house_value')
  ds = tf.data.Dataset.from_tensor_slices((dict(dataframe), labels))
  if shuffle:
    ds = ds.shuffle(buffer_size=len(dataframe))
  ds = ds.batch(batch_size)
  return ds
```

We will wrap the dataframe with tf.data. This will enable us to use feature columns as a bridge to map from the columns in the Pandas dataframe to features used to train the model.

The tf.data API enables you to build complex input pipelines from simple, reusable pieces. For example, the pipeline for an image model might aggregate data from files in a distributed file system, apply random perturbations to each image, and merge randomly selected images into a batch for training. The pipeline for a text model might involve extracting symbols from raw text data, converting them to embedding identifiers with a lookup table, and batching together sequences of different lengths. The tf.data API makes it possible to handle large amounts of data, read from different data formats, and perform complex transformations.

Build an input data pipeline

Numeric columns

The output of a feature column becomes the input to the model. A numeric is the simplest type of column and is used to represent real valued features. When you use this column, your model will receive the column value from the dataframe unchanged.

```
num_c = ['longitude',
         'latitude',
         'housing_median_age',
         'total_rooms',
         'total_bedrooms',
         'population',
         'households',
         'median_income']
```

Most machine learning performance is heavily dependent on the representation of the feature vector. As a result, much of the actual effort in deploying machine learning algorithms goes into the design of preprocessing pipelines and data transformations

Our first step is to build an input data pipeline. We start with numeric columns: the categorical column “ocean_proximity” is not included.

The output of a feature column becomes the input to the model. A numeric is the simplest type of column. It is used to represent real valued features. When using this column, your model will receive the column value from the dataframe unchanged.

Build an input data pipeline

Categorical columns

In this dataset, 'ocean_proximity' is represented as a string. You cannot feed strings directly to a model; instead, you must first map them to numeric values. The categorical vocabulary columns provide a way to represent strings as a one-hot vector.unchanged.

```
cat_i_c = ['ocean_proximity']

for feature_name in cat_i_c:
    vocabulary = dataframe[feature_name].unique()
    cat_c = tf.feature_column.categorical_column_with_vocabulary_list(feature_name, vocabulary)
    one_hot = feature_column.indicator_column(cat_c)
    feature_columns.append(one_hot)
```

In this dataset, 'ocean_proximity' is represented as a string. We cannot feed strings directly to a model. Instead, we must first map them to numeric values. The categorical vocabulary columns provide a way to represent strings as a one-hot vector.unchanged. Use this when your inputs are in string or integer format, and you have an in-memory vocabulary mapping each value to an integer ID.

Build an input data pipeline

Bucketize columns

Often, you don't want to feed a number directly into the model, but instead split its value into different categories based on numerical ranges. Consider the raw data that represents a home's age. Instead of representing the house age as a numeric column, you could split the home age into several buckets using a bucketized column. Notice that the one-hot values below describe which age range each row matches.

```
age_buckets = feature_column.bucketized_column(Age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
feature_columns.append(age_buckets)
```

Often, you don't want to feed a number directly into the model, but instead split its value into different categories based on numerical ranges. Consider our raw data that represents a home's age. Instead of representing the house age as a numeric column, we could split the home age into several buckets using a bucketized column. Notice the one-hot values below describe which age range each row matches.

Build an input data pipeline

Feature cross columns

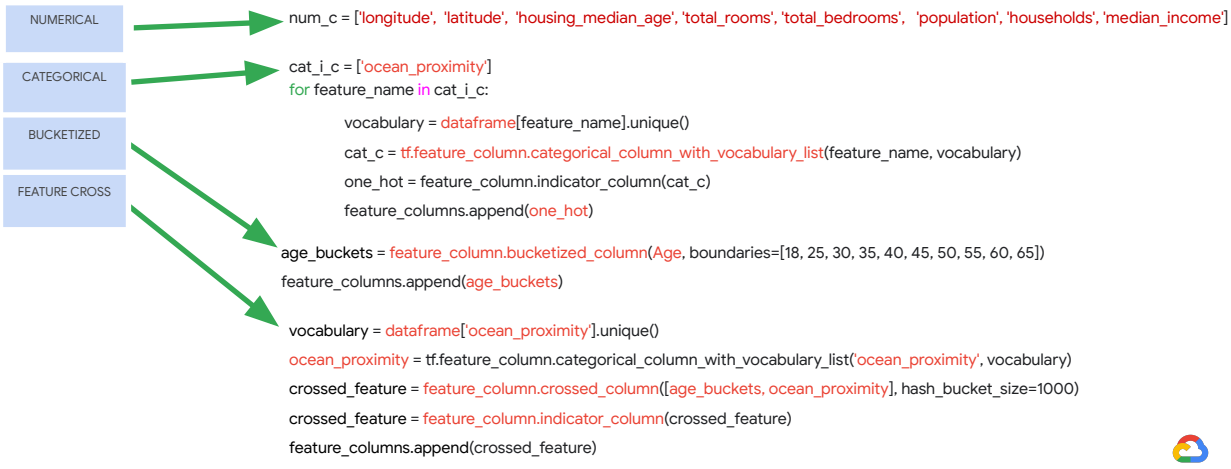
Combining features into a single feature, better known as a feature cross, enables a model to learn separate weights for each combination of features.

```
vocabulary = dataframe['ocean_proximity'].unique()
ocean_proximity =
tf.feature_column.categorical_column_with_vocabulary_list('ocean_proximity', vocabulary)

crossed_feature = feature_column.crossed_column([age_buckets, ocean_proximity],
hash_bucket_size=1000)
crossed_feature = feature_column.indicator_column(crossed_feature)
feature_columns.append(crossed_feature)
```

Combining features into a single feature, better known as feature crosses, enables a model to learn separate weights for each combination of features.

https://www.tensorflow.org/api_docs/python/tf/feature_column/crossed_column?version=stable

Feature-engineered input data pipeline



Here is our feature-engineered input data pipeline.

Build the DNN

Keras Sequential Model API

```
feature_layer = tf.keras.layers.DenseFeatures(feature_columns)
model = tf.keras.Sequential([
  feature_layer,
  layers.Dense(12,  input_dim=8, activation='relu'),
  layers.Dense(8, activation='relu'),
  layers.Dense(1, activation='linear', name='median_house_value')
])
model.compile(optimizer='adam',
              loss='mse',
              metrics=['mse'])
history = model.fit(train_ds,
          validation_data=val_ds,
          epochs=32)
```

Here, use the Keras Sequential Model API to build the deep neural network. The Sequential Model API is a way of creating deep learning models where an instance of the Sequential class is created and model layers are created and added to it.

Some common and useful layer types you can choose from are:

- **Dense**: Fully connected layer and the most common type of layer used on multi-layer perceptron models.
- **Dropout**: Apply dropout to the model, setting a fraction of inputs to zero in an effort to reduce overfitting.
- **Merge**: Combine the inputs from multiple models into a single model.

In this model we build a linear stack of dense layers. The first layer in your model must specify the shape of the input. This is the number of input attributes and is defined by the input_dim argument. This argument expects an integer.

Keras also supports a range of standard neuron activation functions, such as relu, softmax, rectifier, tanh and sigmoid. You typically specify the type of activation function used by a layer in the activation argument, which takes a string value. The last layer is the output layer, and we use “linear” as the activation function. The name of our label (or what we are predicting) is the ‘median_house_value.’

After you define your model, it needs to be compiled. This creates the efficient

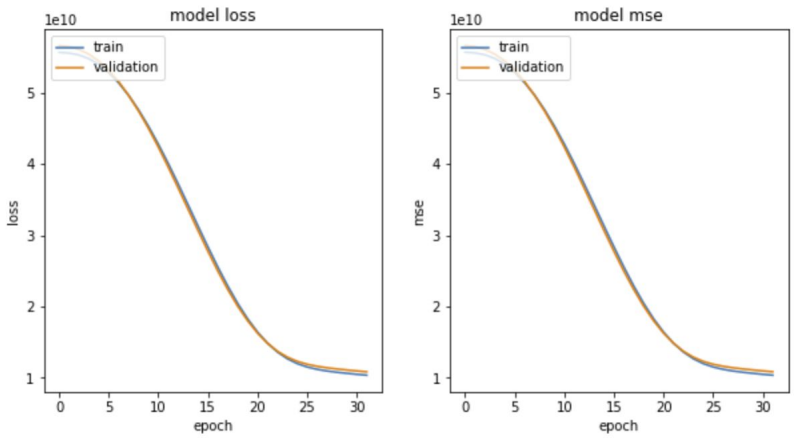
your model during training. You compile your model using the compile() function, and it accepts three important attributes: (1) Model optimizer; (2) Loss function; and (3) Metrics.

Recall that the optimizer is the search technique used to update weights in your model. Adaptive Moment Estimation (Adam) is a popular gradient descent optimizer that uses adaptive learning rates, so we use that.

Recall that the loss function is the evaluation of the model used by the optimizer to navigate the weight space. We choose the mean squared error. Metrics are evaluated by the model during training. We choose mse. Note that we declared batch_size (that is, the number of training instances shown to the model before a weight update is performed) when we initialized the training dataset.

Finally, the model is trained using the model.fit() function. An epoch is the number of times that the model is exposed to the training dataset. Fitting the model returns a history object with details and metrics calculated for the model each epoch. We use this for graphing model performance.

Model loss and MSE



Next, we use matplotlib to draw the model's loss curves for training and validation. A line plot is also created showing the mean squared error loss over the training epochs for both the train (blue) and test (orange) sets.

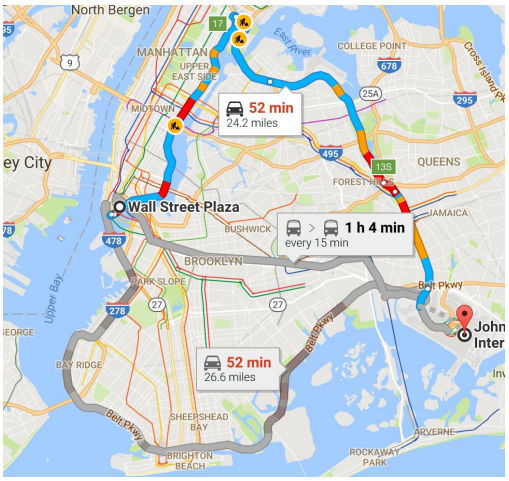
Model prediction

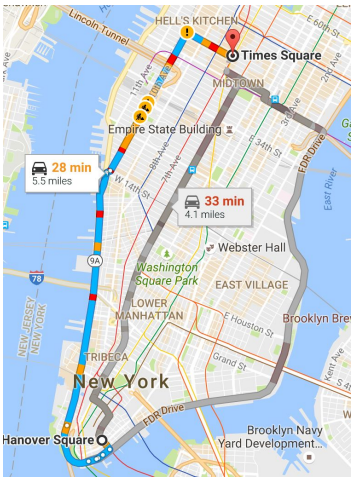
House near the ocean

```
Median_house_value is $249,000, prediction is $234,000

model.predict({
    'longitude': tf.convert_to_tensor([-122.43]),
    'latitude': tf.convert_to_tensor([37.63]),
    'housing_median_age': tf.convert_to_tensor([34.0]),
    'total_rooms': tf.convert_to_tensor([4135.0]),
    'total_bedrooms': tf.convert_to_tensor([687.0]),
    'population': tf.convert_to_tensor([2154.0]),
    'households': tf.convert_to_tensor([742.0    ]),
    'median_income': tf.convert_to_tensor([ 4.9732]),
    'ocean_proximity': tf.convert_to_tensor(['NEAR OCEAN'])

}, steps=1)
```

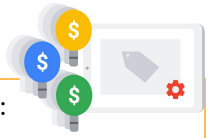
Once we have incorporated our features and fit our model, we will perform a prediction. Here, we are interested in seeing how well our model performs against test data for a house with a median value of $249,000 with the longitude and latitudes listed here, as well as the feature values shown. Our prediction shows $234,000. Yours may vary due to the random shuffling of the data.

ML problem: Estimating taxi fare





http://www.nyc.gov/html/tlc/html/passenger/taxicab_rate.shtml



Taxi fare:

Fare amount
+
Distance travelled
+
Number of passengers
+
Pickup and dropoff location
+
Pickup date and time

Our next machine learning problem is to predict taxi fare price in New York city using Keras.

Define features, default values, and label

```
CSV_COLUMNS = [
    'fare_amount',
    'pickup_datetime',
    'pickup_longitude',
    'pickup_latitude',
    'dropoff_longitude',
    'dropoff_latitude',
    'passenger_count',
    'key',
]
LABEL_COLUMN = 'fare_amount'
STRING_COLS = ['pickup_datetime']
NUMERIC_COLS = ['pickup_longitude', 'pickup_latitude',
                'dropoff_longitude', 'dropoff_latitude',
                'passenger_count']
DEFAULTS = [[0.0], ['na'], [0.0], [0.0], [0.0], [0.0], [0.0], ['na']]
DAYS = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

These next slides will highlight the key concepts you will see when completing the labs.

Here is step one of a two-step process to build the pipeline. Step one is to define our columns of data, which column we're predicting for, and the default values.

In our lab, we'll first define our columns of data. Here we define three variables called CSV_COLUMNS, LABEL_COLUMN, and DEFAULTS.

Create an input pipeline using tf.data

```
def features_and_labels(row_data):
    for unwanted_col in ['pickup_datetime', 'key']:
        row_data.pop(unwanted_col)
    label = row_data.pop(LABEL_COLUMN)
    return row_data, label  # features, label

# load the training data
def load_dataset(pattern, batch_size=1, mode=tf.estimator.ModeKeys.EVAL):
  dataset = (tf.data.experimental.make_csv_dataset(pattern, batch_size,
CSV_COLUMNS, DEFAULTS)
             .map(features_and_labels) # features, label
             )
  if mode == tf.estimator.ModeKeys.TRAIN:
        dataset = dataset.shuffle(1000).repeat()
  dataset = dataset.prefetch(1)
  return dataset
```

Step 2 is to define two functions: a function to define the features and label we want to use and a function to load the training data.

Build the DNN

Keras Functional Model API

```
def rmse(y_true, y_pred):  # Root mean square error
    return tf.sqrt(tf.reduce_mean(tf.square(y_pred - y_true)))

def build_dnn_model():  # Build the DNN model
    # input layer
    inputs = {
        colname: tf.keras.layers.Input(name=colname, shape=(),
dtype='float32')
        for colname in NUMERIC_COLS
    }

# feature_columns
    feature_columns = {
        colname: tf.feature_column.numeric_column(colname)
        for colname in NUMERIC_COLS
    }
```

Here we define a function to build the deep neural network model.

Build the DNN

Keras Functional Model API

```
# Constructor for DenseFeatures takes a list of numeric columns
# The Functional API in Keras requires you specify:
LayerConstructor()(inputs)

dnn_inputs = tf.keras.layers.DenseFeatures(feature_columns.values())(inputs)

# two hidden layers of [32, 8]
h1 = tf.keras.layers.Dense(32, activation='relu', name='h1')(dnn_inputs)
h2 = tf.keras.layers.Dense(8, activation='relu', name='h2')(h1)

# final output is a linear activation because this is regression
output = tf.keras.layers.Dense(1, activation='linear', name='fare')(h2)
model = tf.keras.models.Model(inputs, output)

# compile model
model.compile(optimizer='adam', loss='mse', metrics=[rmse, 'mse'])
return model


print("Here is our DNN architecture so far:\n")
model = build_dnn_model()
print(model.summary())
```

Here, use the Keras Functional Model API to build the deep neural network. The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple inputs or outputs. The functional API in Keras is an alternative way of creating models that offers a lot more flexibility, including creating more complex models.

You'll notice that we are still using the same activation functions, e.g. "relu" for our hidden layers and "linear" for the final output layer. Similarly, our model optimizer, loss, and metrics remain the same.

INSTRUCTOR NOTE: Source:
https://machinelearningmastery.com/keras-functional-api-deep-learning/
More about the functional model:
The Keras functional API provides a more flexible way for defining models. It specifically allows you to define multiple input or output models and models that share layers. More than that, it allows you to define ad hoc acyclic network graphs.

Models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a model that specifies the layers to act as the input and output to the model.

Unlike the Sequential model, you must create and define a standalone input layer that specifies the shape of input data. The input layer takes a shape argument that is a tuple that indicates the dimensionality of the input data. When input data is one-dimensional, such as for a multilayer Perceptron, the shape must explicitly leave

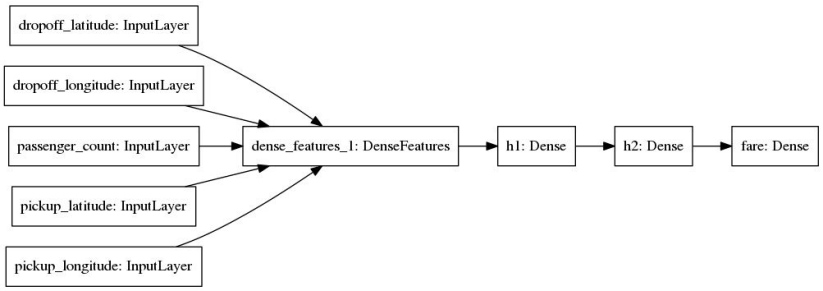
room for the shape of the mini-batch size used when splitting the data when training the network.

The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A bracket notation is used, such that after the layer is created, the layer that the input to the current layer comes from is specified.

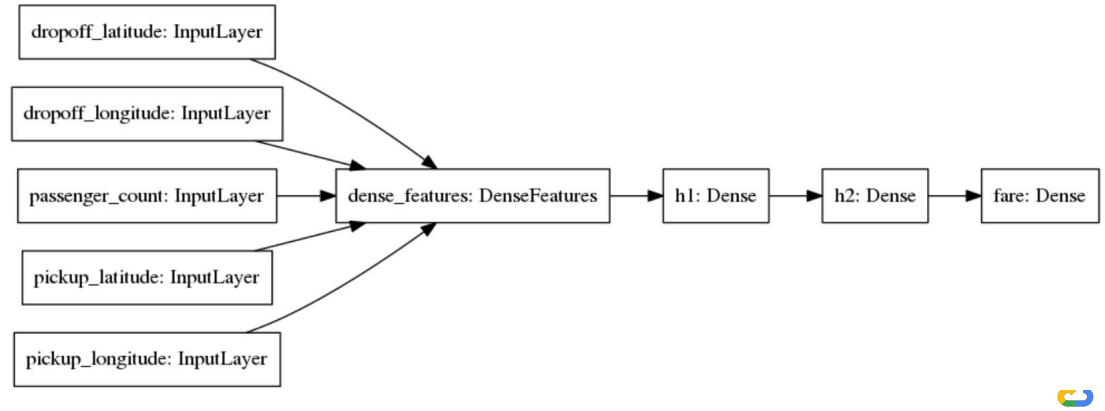
Summary

Connect Dense Features layer to rest of model

```
dnn_inputs = tf.keras.layers.DenseFeatures(feature_columns)(inputs)
h1 = tf.keras.layers.Dense(32, activation='relu', name='h1')(dnn_inputs)
h2 = tf.keras.layers.Dense(8, activation='relu', name='h2')(h1)
output = tf.keras.layers.Dense(1, activation='linear', name='fare')(h2)
model = tf.keras.models.Model(inputs, output)
```



Visualize the model



After defining our features, default values, and label, we visualize the model.

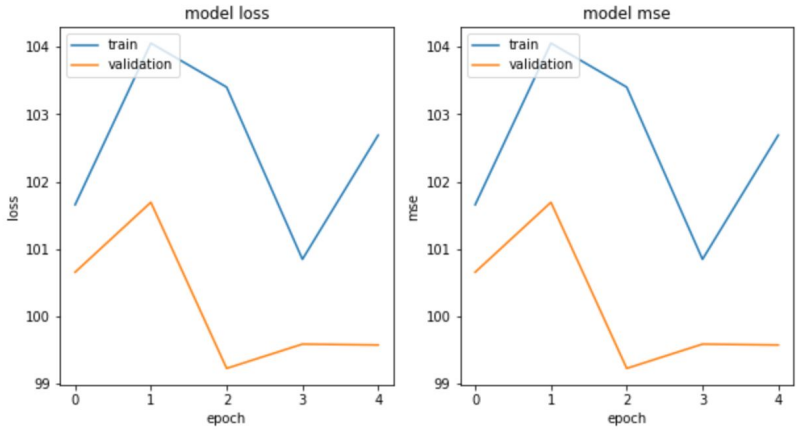
Train the model

```
TRAIN_BATCH_SIZE = 32
NUM_TRAIN_EXAMPLES = 59621 * 5
NUM_EVALS = 5
NUM_EVAL_EXAMPLES = 14906
```

```
trainds = load_dataset('../data/taxi-train*',
                       TRAIN_BATCH_SIZE,
                       tf.estimator.ModeKeys.TRAIN)
evalds = load_dataset('../data/taxi-valid*',
                      1000,
                      tf.estimator.ModeKeys.EVAL).take(NUM_EVAL_EXAMPLES//1000)

steps_per_epoch = NUM_TRAIN_EXAMPLES // (TRAIN_BATCH_SIZE * NUM_EVALS)

history = model.fit(trainds,
                    validation_data=evalds,
                    epochs=NUM_EVALS,
                    steps_per_epoch=steps_per_epoch)
```

Here we train the model, initializing variables such as train_batch_size and num_train_examples.
tf.estimator.ModeKeys.

Model loss and MSE



Next, we use matplotlib to draw the model's loss curves for training and validation. A line plot is also created showing the mean squared error loss over the training epochs for both the train (blue) and test (orange) sets.

Model prediction

New York taxi fare at these coordinates is $12.29

```
model.predict({
    'pickup_longitude': tf.convert_to_tensor([-73.982683]),
    'pickup_latitude': tf.convert_to_tensor([40.742104]),
    'dropoff_longitude': tf.convert_to_tensor([-73.983766]),
    'dropoff_latitude': tf.convert_to_tensor([40.755174]),
    'passenger_count': tf.convert_to_tensor([3.0]),
    'pickup_datetime': tf.convert_to_tensor(['2019-06-03 04:21:29
 UTC'], dtype=tf.string),
}, steps=1)
```

After we incorporate our features and fit our model, we perform a prediction. Here, we are interested in seeing how well our model performs against test data for a taxi ride in New York City. Our prediction shows $12.29. Yours may vary due to the random shuffling of the data.

Temporal features

```
# TODO 1a
def parse_datetime(s):
    if type(s) is not str:
        s = s.numpy().decode('utf-8')
    return datetime.datetime.strptime(s, "%Y-%m-%d %H:%M:%S %Z")


# TODO 1b
def get_dayofweek(s):
    ts = parse_datetime(s)
    return DAYS[ts.weekday()]


# TODO 1c
@tf.function
def dayofweek(ts_in):
    return tf.map_fn(
        lambda s: tf.py_function(get_dayofweek, inp=[s], Tout=tf.string),
        ts_in)
```

Here we define three functions:

1. The first is the parse the date_time feature column, where we use numpy().`decode('UTF-8')` to decode the numpy array of string elements. The **datetime.strptime()** Python class method creates a **datetime** object from a string representing a date and time and a corresponding format string, in our case year, month day, hour, minute, second and time zone.
2. The second is the get_dayofweek function, where we get the days of the week from the parse_datetime function and return DAYS (a variable we initialized earlier).
3. Next we have the dayofweek function. We pass in the get_dayofweek tf.string parameter to a tf.py_function, which then maps a string to a tensor, so that every element in dayofweek is passed to the model as a tensor.

INSTRUCTOR NOTES:
`tf.py_function` makes it possible to express control flow using Python constructs (`if`, `while`, `for`, etc.), instead of TensorFlow control flow constructs (`tf.cond`, `tf.while_loop`).

Lambda layer - The `Lambda` layer exists so that arbitrary TensorFlow functions can be used when constructing `Sequential` and Functional API models.

`Lambda` layers are best suited for simple operations or quick experimentation. For more advanced usecases, follow this guide for subclassing `tf.keras.layers.Layer`.

tf.map_fun - The simplest version of `map_fn` repeatedly applies the callable `fn` to a sequence of elements from first to last. The elements are made of the tensors unpacked from `elems`. `dtype` is the data type of the return value of `fn`. Users must provide `dtype` if it is different from the data type of `elems`. It takes one element of each input tensor at a time (so a tuple of two elements) and applies the given function to it, which should return the same structure that you passed in `dtypes` (a tuple of two elements, too); if you don't give a `dtypes`, then the expected output is the same as the input (again, a tuple of two elements, so in your case `dtypes` is optional).

Computing Euclidean distance

```
# TODO 2
def euclidean(params):
    lon1, lat1, lon2, lat2 = params
    londiff = lon2 - lon1
    latdiff = lat2 - lat1
    return tf.sqrt(londiff*londiff + latdiff*latdiff)
```

Max/Min scaling latitude and longitude

```
def scale_longitude(lon_column):
    return (lon_column + 78)/8.
```

```
def scale_latitude(lat_column):
    return (lat_column - 37)/8.
```

Scaling latitude and longitude

It is very important for numerical variables to get scaled before they are "fed" into the neural network. Here we use min-max scaling (also called normalization) on the geolocation fetures. Later in our model, you will see that these values are shifted and rescaled so that they end up ranging from 0 to 1.

First, we create a function named 'scale_longitude', where we pass in all the longitudinal values and add 78 to each value. Note that our scaling longitude ranges from -70 to -78. Thus, the value 78 is the maximum longitudinal value. The delta or difference between -70 and -78 is 8. We add 78 to each longitidunal value and then divide by 8 to return a scaled value.

Next, we create a function named 'scale_latitude,' where we pass in all the latitudinal values and subtract 37 from each value. Note that our scaling longitude ranges from -37 to -45. Thus, the value 37 is the minimal latitudinal value. The delta or difference between -37 and -45 is 8. We subtract 37 from each latitudinal value and then divide by 8 to return a scaled value.

The pickup/dropoff longitude and latitude data are crucial to predicting the fare amount because fare amounts in NYC taxis are largely determined by the distance traveled. As such, we need to teach the model the Euclidean distance between the pickup and dropoff points.

Recall that latitude and longitude allows us to specify any location on Earth using a set of coordinates. In our training data set, we restricted our data points to only pickups and dropoffs within NYC. New York city has an approximate longitude range of -74.05 to -73.75 and a latitude range of 40.63 to 40.85.

Computing Euclidean distance
The dataset contains information regarding the pickup and dropoff coordinates. However, there is no information regarding the distance between the pickup and dropoff points. Therefore, we create a new feature that calculates the distance between each pair of pickup and dropoff points. We can do this using the Euclidean Distance, which is the straight-line distance between any two coordinate points.

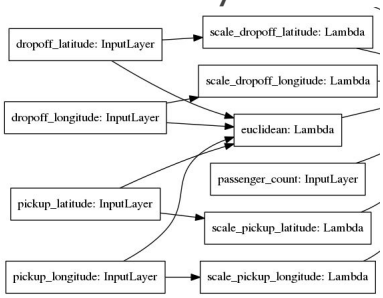
Geolocation features

```
# Scaling longitude from range [-70, -78] to [0, 1]
for lon_col in ['pickup_longitude', 'dropoff_longitude']:
    transformed[lon_col] = layers.Lambda(
        scale_longitude,
        name="scale_{}".format(lon_col))(inputs[lon_col])

# Scaling latitude from range [37, 45] to [0, 1]
for lat_col in ['pickup_latitude', 'dropoff_latitude']:
    transformed[lat_col] = layers.Lambda(
        scale_latitude,
        name='scale_{}'.format(lat_col))(inputs[lat_col])

# TODO 2
# add Euclidean distance
transformed['euclidean'] = layers.Lambda(
    euclidean,
    name='euclidean')([inputs['pickup_longitude'],
                       inputs['pickup_latitude'],
                       inputs['dropoff_longitude'],
                       inputs['dropoff_latitude']])
feature_columns['euclidean'] = fc.numeric_column('euclidean')
```

Here are the transformations we implement on the geolocation features.

Create new features using Lambda Layers

```
def euclidean(params):
    lon1, lat1, lon2, lat2 = params
    londiff = lon2 - lon1
    latdiff = lat2 - lat1
    return tf.sqrt(londiff*londiff + latdiff*latdiff)

...
```




```
    transformed['euclidean'] = tf.keras.layers.Lambda(euclidean, name='euclidean')([
        inputs['pickup_longitude'],
        inputs['pickup_latitude'],
        inputs['dropoff_longitude'],
        inputs['dropoff_latitude']
    ])
    feature_columns['euclidean'] = tf.feature_column.numeric_column('euclidean')
```

Example where we created new features using Lambda Layers. ***Normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.*** **Keras** employs a similar naming scheme to define anonymous/custom layers. Lambda layers in Keras help you to implement layers or functionality that are not prebuilt and **which do not require trainable weights**.

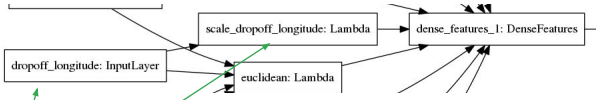
Source: https://www.quora.com/What-is-the-Lambda-layer-in-Keras

Geolocation bucketized features

```
if True:
        # https://buganizer.corp.google.com/issues/135479527
        # featurecross lat, lon into nxn buckets, then embed
        nbuckets = NBUCKETS
        latbuckets = np.linspace(0, 1, nbuckets).tolist()
        lonbuckets = np.linspace(0, 1, nbuckets).tolist()
        b_plat = tf.feature_column.bucketized_column(feature_columns['pickup_latitude'], latbuckets)
        b_dlat = tf.feature_column.bucketized_column(feature_columns['dropoff_latitude'], latbuckets)
        b_plon = tf.feature_column.bucketized_column(feature_columns['pickup_longitude'], lonbuckets)
        b_dlon = tf.feature_column.bucketized_column(feature_columns['dropoff_longitude'], lonbuckets)
        ploc = tf.feature_column.crossed_column([b_plat, b_plon], nbuckets * nbuckets)
        dloc = tf.feature_column.crossed_column([b_dlat, b_dlon], nbuckets * nbuckets)
        pd_pair = tf.feature_column.crossed_column([ploc, dloc], nbuckets ** 4)
        feature_columns['pickup_and_dropoff'] = tf.feature_column.embedding_column(pd_pair, 100)
```

Here are the continued transformations we implement on the geolocation features.

Can replace feature columns by scaled values




```
for lon_col in ['pickup_longitude', 'dropoff_longitude']:  # in range -70 to -78
        transformed[lon_col] = tf.keras.layers.Lambda(
            lambda x: (x+78)/8.0,
            name='scale_{}'.format(lon_col)
        )(inputs[lon_col])
```

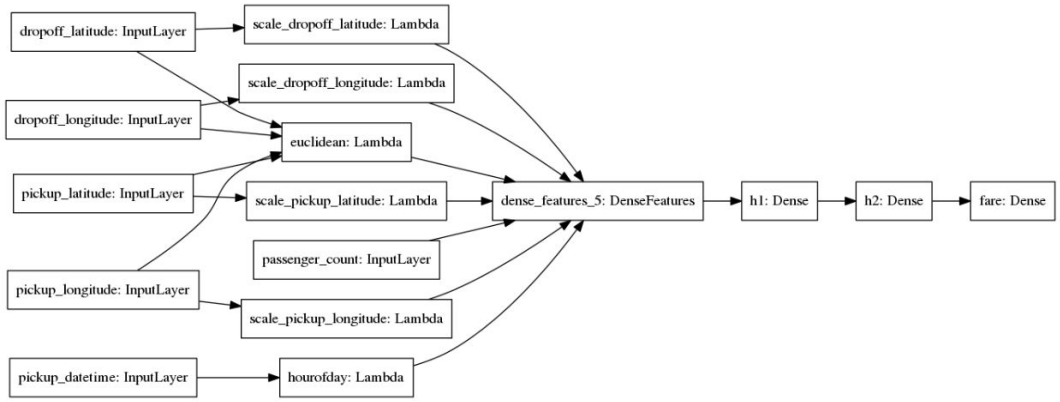
Example where we replaced features columns by scaled values.

Feature cross temporal features

```
# hour of day from timestamp of form '2010-02-08 09:17:00+00:00'
    if True:
        transformed['hourofday'] = tf.keras.layers.Lambda(
            lambda x: tf.strings.to_number(tf.strings.substr(x, 11, 2), out_type=tf.dtypes.int32),
            name='hourofday'
        )(inputs['pickup_datetime'])
        feature_columns['hourofday'] = tf.feature_column.indicator_column(
            tf.feature_column.categorical_column_with_identity('hourofday', num_buckets=24))

    if False:
        # https://buganizer.corp.google.com/issues/137795281
        # day of week is hard because there is no TensorFlow function for date handling
        transformed['dayofweek'] = tf.keras.layers.Lambda(
            lambda x: dayofweek(x),
            name='dayofweek_pyfun'
        )(inputs['pickup_datetime'])
        transformed['dayofweek'] = tf.keras.layers.Reshape((),
name='dayofweek')(transformed['dayofweek'])
        feature_columns['dayofweek'] = tf.feature_column.indicator_column(
            tf.feature_column.categorical_column_with_vocabulary_list(
              'dayofweek', vocabulary_list=DAYS))
```

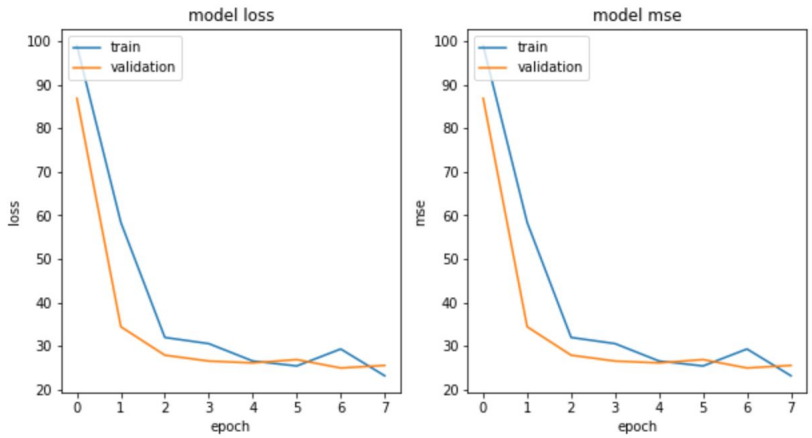
Here are the continued transformations we implement on the temporal features.

Visualize the model



Model visualization showing all engineered features.

Model loss and MSE



Next, we use matplotlib to draw the model's loss curves for training and validation. A line plot is also created showing the mean squared error loss over the training epochs for both the train (blue) and test (orange) sets.

Here's a common loss curve shape -- the loss drops off rapidly with your big steps down the gradient and then smooths out over time with smaller steps as it reaches a minima on the loss surface.

Model prediction

New York taxi fare at these coordinates is $7.28

```
model.predict({
    'pickup_longitude': tf.convert_to_tensor([-73.982683]),
    'pickup_latitude': tf.convert_to_tensor([40.742104]),
    'dropoff_longitude': tf.convert_to_tensor([-73.983766]),
    'dropoff_latitude': tf.convert_to_tensor([40.755174]),
    'passenger_count': tf.convert_to_tensor([3.0]),
    'pickup_datetime': tf.convert_to_tensor(['2019-06-03 04:21:29
 UTC'], dtype=tf.string),
}, steps=1)
```

After we incorporate our features and fit our model, we perform a prediction. Here, we are interested in seeing how well our model performs against test data for a taxi ride in New York City. Our prediction shows $7.28. Yours may vary due to the random shuffling of the data.

RMSE: Summary table

Model	Taxi Fare	Description
Baseline	12.29	Baseline model - no feature engineering
Feature Engineered	07.28	Feature Engineered Model

Here is a summary of the RMSE values.

Lab: Basic Feature Engineering in Keras

1. Create an input pipeline using tf.data.
2. Engineer features to create categorical, crossed, and numerical feature columns.

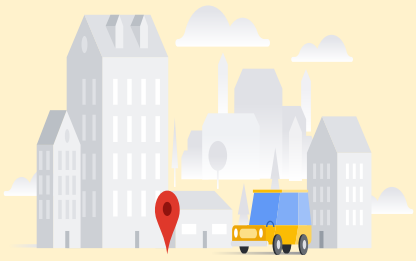
The objectives of the first lab are to:

1. Create an input pipeline using tf.data.
2. Engineer features to create categorical, crossed, and numerical feature columns.

Link to lab: [ML on GCP C4] Basic Feature Engineering in Keras (qwiklabs). Github repo here. CBL197

Lab: Adv. Feature Engineering in Keras

1. Process temporal feature columns in Keras.
2. Use Lambda layers to perform feature engineering on geolocation features.
3. Create bucketized and crossed feature columns.

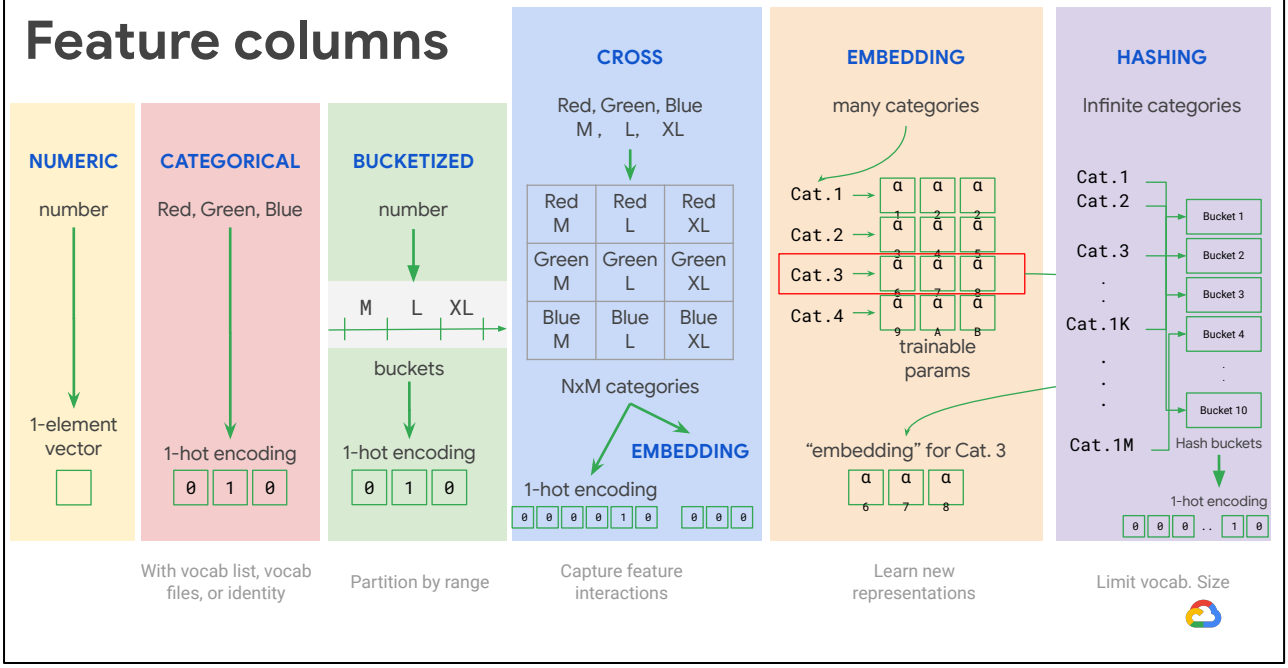


The objectives of the first lab are to:

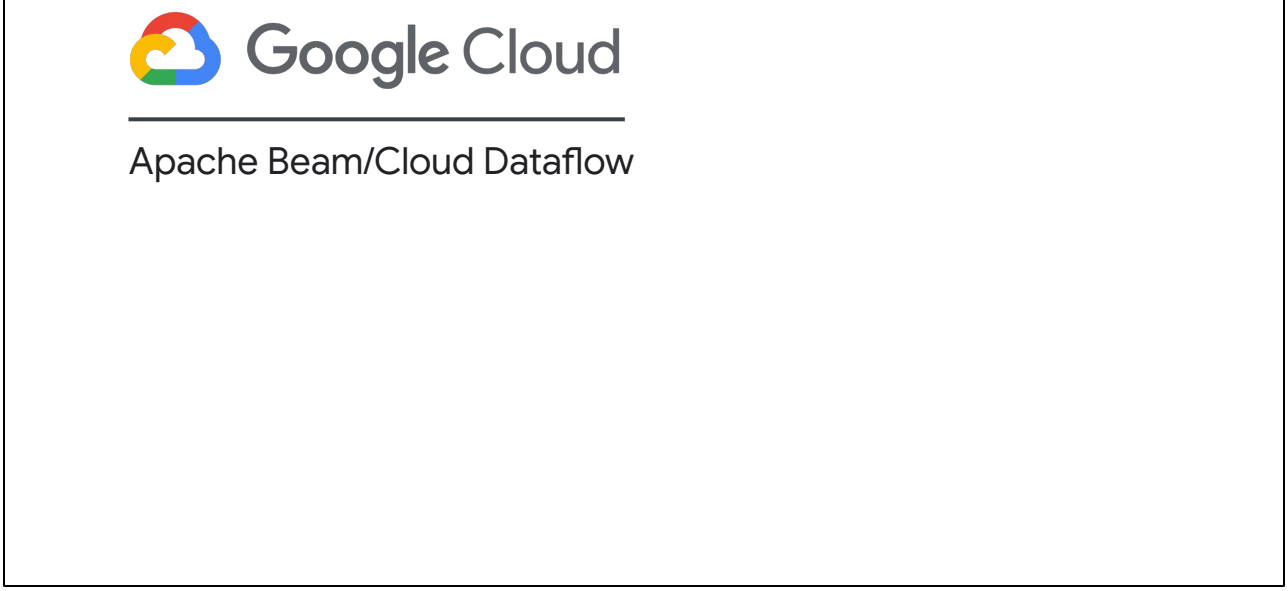
1. Define features, the Label, and default values.
2. Create an input pipeline using tf.data.
3. Create ML models using Keras.
4. Visualize the DNN model layers.
5. Train the model/visualize the loss curve.
6. Incorporate temporal features.
7. Incorporate Geolocation Features with Bucketization and Feature Crosses.
8. Feature Cross Temporal Features.
9. Create and test a prediction model.

Link to lab:

[ML on GCP C4] Performing Advanced Feature Engineering in Keras_ github repo here.
CBL198



This slide summarizes the different feature engineering types that can be applied to feature columns. We will explore them in the next four labs.



In the next part of this section you will learn more about Google Cloud Dataflow which is a complementary technology to Apache Beam and both of them can help you build and run pre-processing and feature engineering.

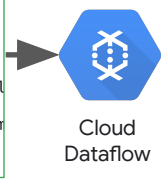
Beam is a way to write elastic data processing pipelines

Cloud Dataflow

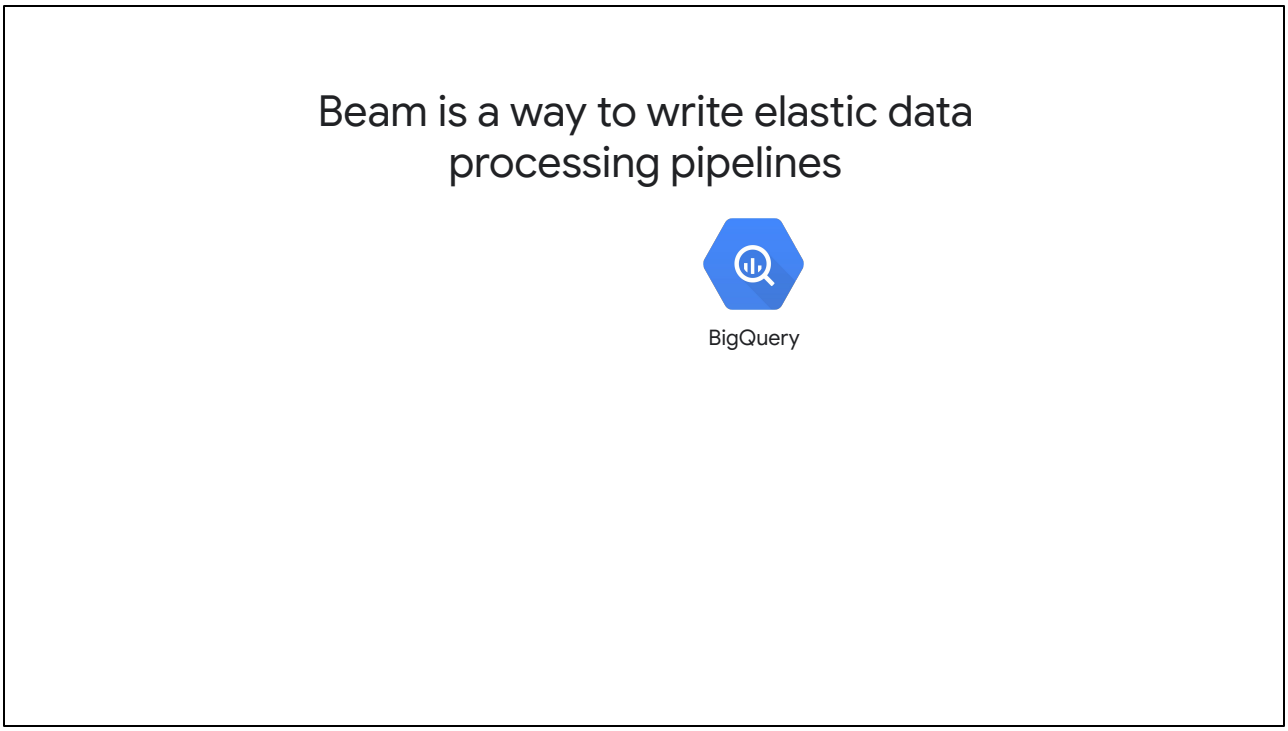
So first of all. What is Cloud Dataflow? One way to think about feature pre-processing or even any data transformation is to think in terms of pipelines. Here, when I say, pipeline, I mean,

Beam is a way to write elastic data processing pipelines

```
def packageHelp(record, keyword):
    count=0
    package_name=''
    if record is not None:
      lines=record.split('\n')
      for line in lines:
        if line.startswith(keyword):
          package_name=line
        if 'FIXME' in line or 'TODO' in l
          count+=1
      packages = (getPackages(package_nam
      for p in packages:
          yield (p,count)
```



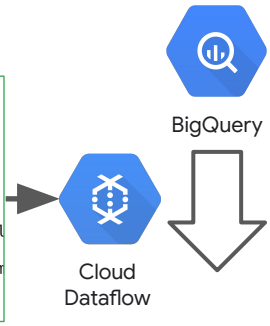
a **sequence of steps that change data from one format into another**. So suppose you have some data in a data warehouse,

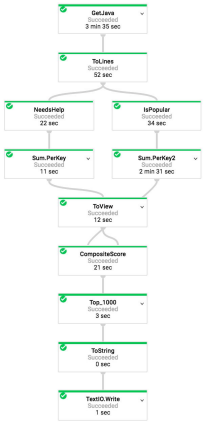


like BigQuery, then you can use BigQuery as an input to your pipeline,

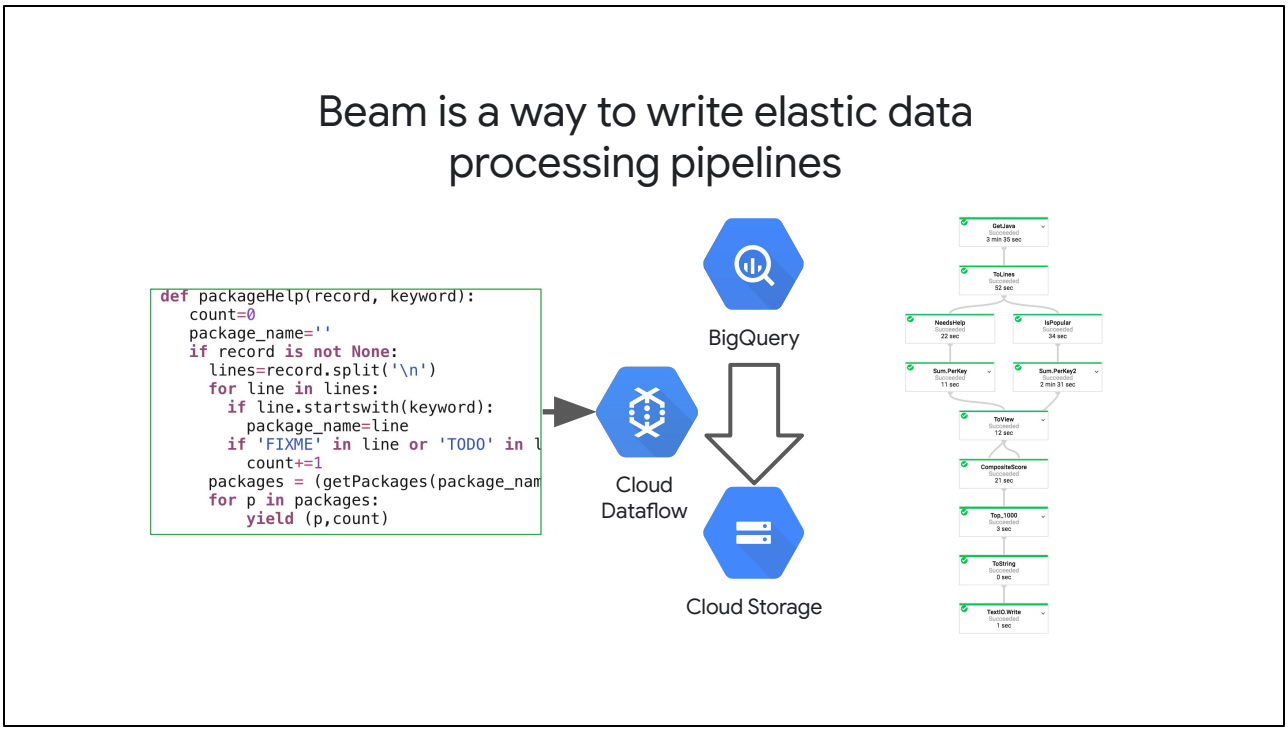
Beam is a way to write elastic data processing pipelines

```
def packageHelp(record, keyword):
    count=0
    package_name=''
    if record is not None:
      lines=record.split('\n')
      for line in lines:
        if line.startswith(keyword):
          package_name=line
        if 'FIXME' in line or 'TODO' in l
          count+=1
      packages = (getPackages(package_nam
      for p in packages:
          yield (p,count)
```





do a sequence of steps to transform the data, maybe introduce some new features as part of the transformation.



Finally you can save the result to an output like Google Cloud Storage.

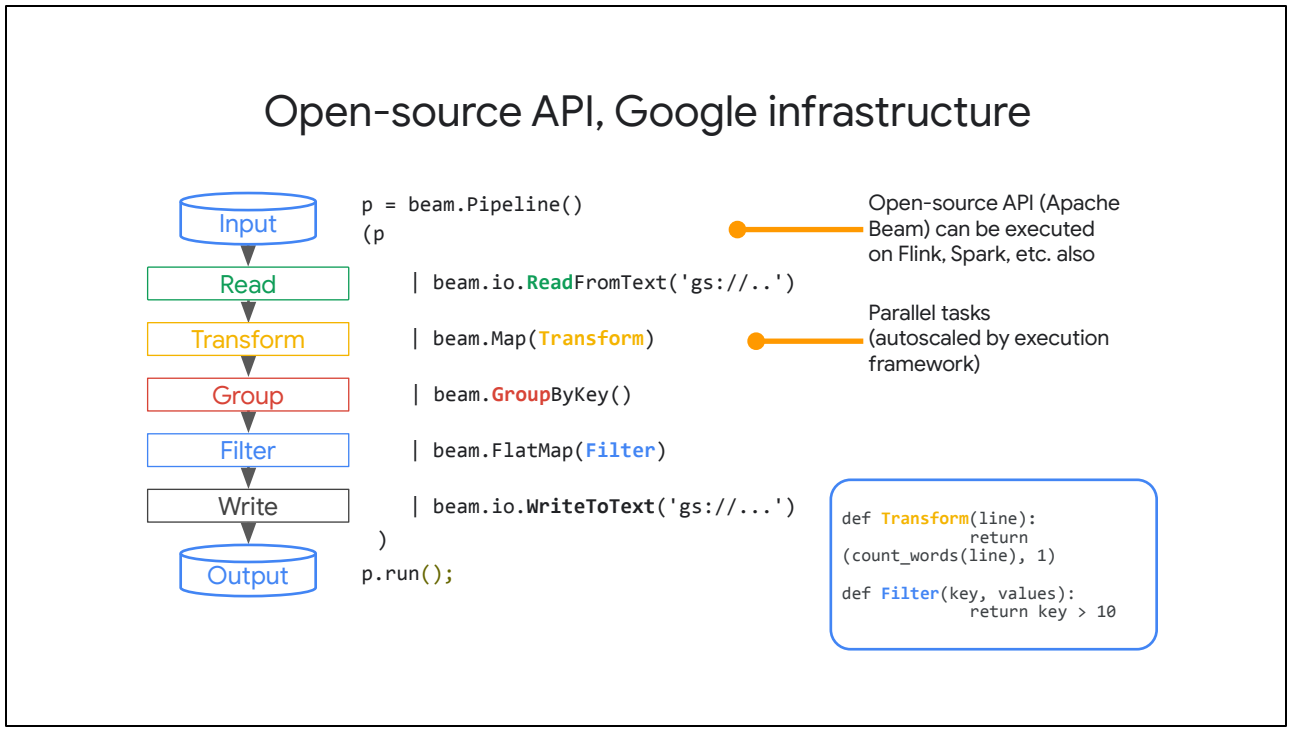
Now Google Cloud Dataflow is a platform that allows you to run these kinds of data processing pipelines. Dataflow can run pipelines written in Python and Java programming languages. Dataflow sets itself a part as a platform for data transformations because it is a serverless, a fully managed offering from Google that allows you to execute data processing pipelines at scale. As a developer you don't have to worry about managing the size of the cluster that runs your pipeline. Dataflow can change the amount of compute resources, the number of servers that will run your pipeline, elastically, depending on the amount of data that your pipeline needs to process.
</KMO>

Notes:

The idea is to write Java code, deploy it to Dataflow which then executes the pipeline.

The pipeline here reads data from BigQuery, does a bunch of processing and writes its output to CloudStorage.

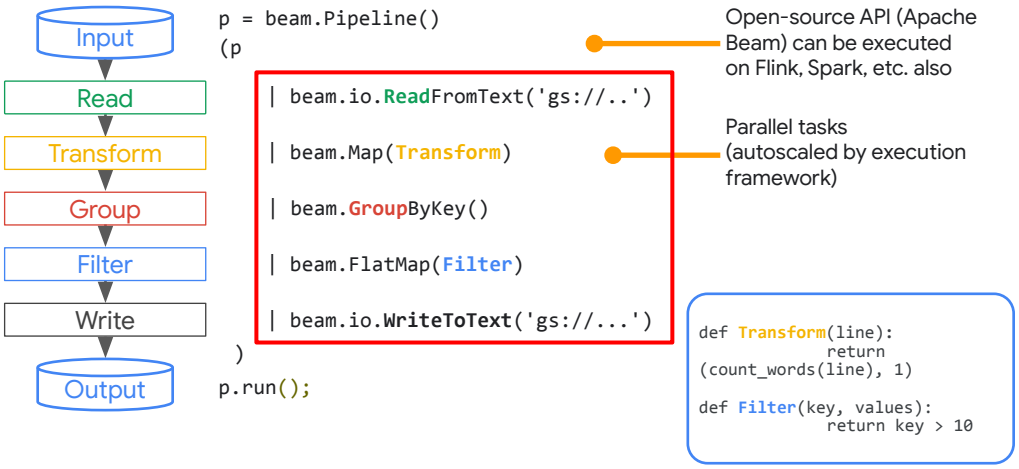
Elastic: unlike Dataproc, there is no need to launch a cluster. Like BigQuery in that respect.



The way that you write code for Dataflow is by using a open-source library called Apache Beam. So, to implement a data processing pipeline, you write your code using the Apache Beam APIs and then deploy the code to Cloud Dataflow.

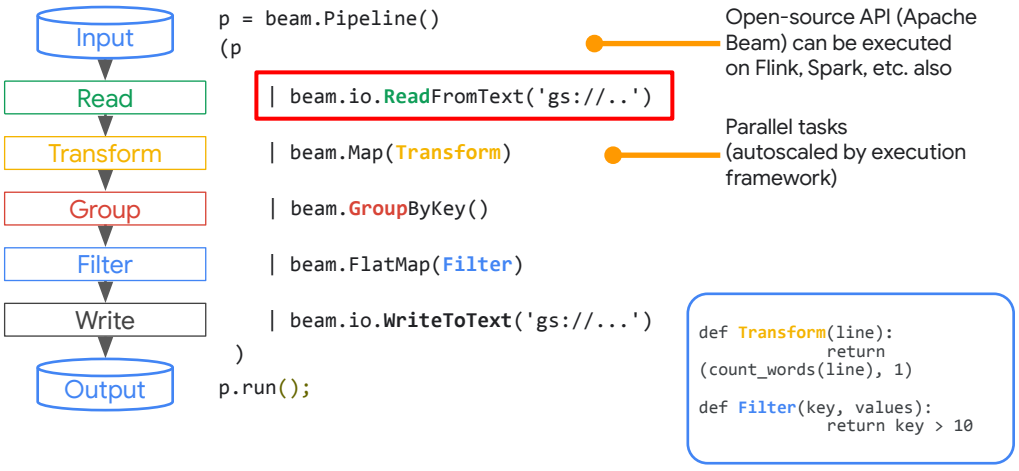
One thing that makes Apache Beam easy to use is that the code written for Beam is similar to how people think of data processing pipelines.

Open-source API, Google infrastructure

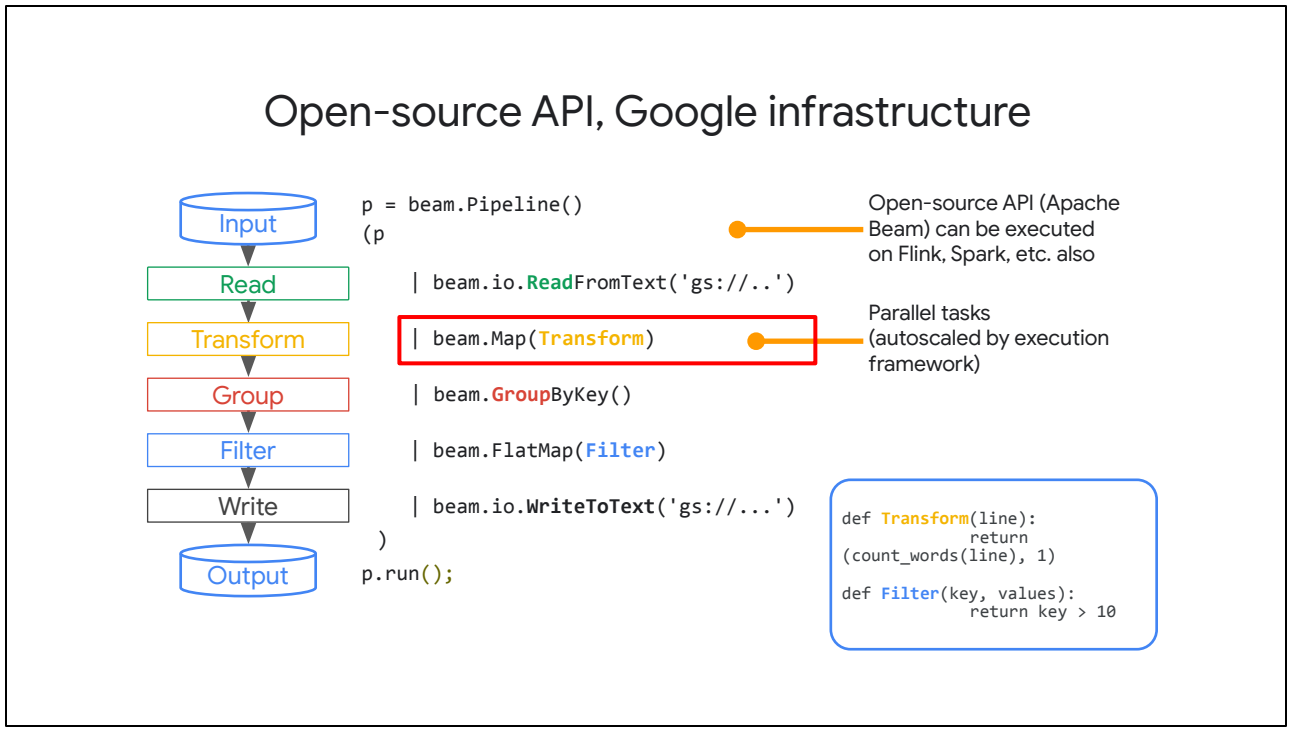


Take a look at the pipeline in the center of the slide. This sample Python code analyzes the number of words in lines of text in documents.

Open-source API, Google infrastructure

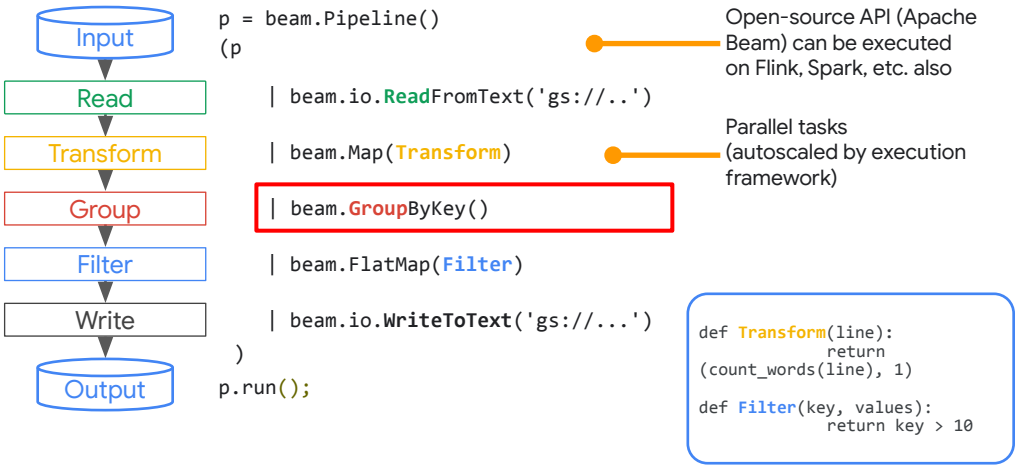


So, as an input to the pipeline you may want to read text files from Google Cloud Storage.



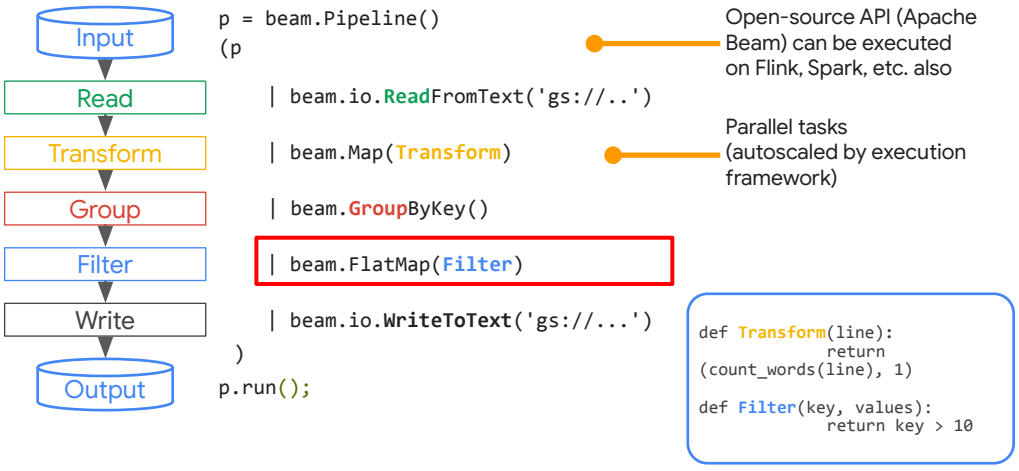
Then, you transform the data, figure out the number of words in each line of text. As I will explain shortly, this kind of a transformation can be automatically scaled by Dataflow to run in parallel.

Open-source API, Google infrastructure

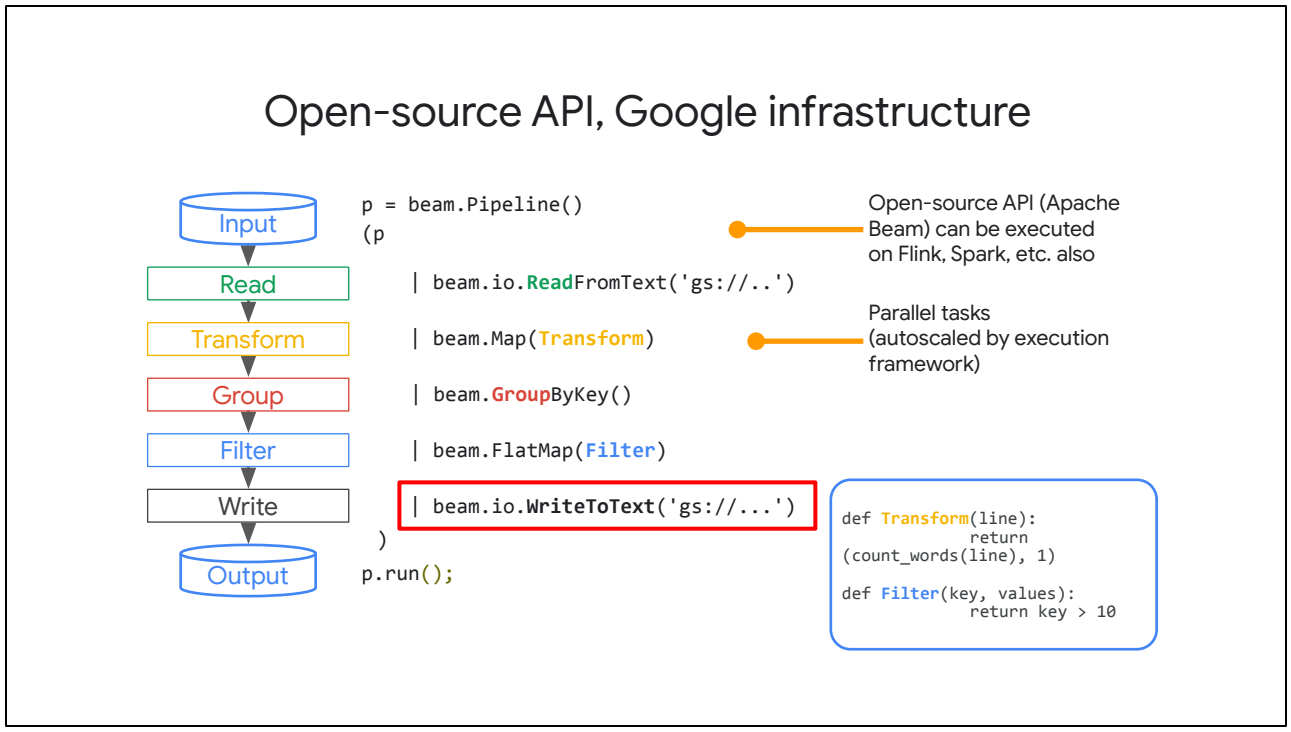


Next, In your pipeline, you can group lines by the number of words using grouping and other aggregation operations.

Open-source API, Google infrastructure

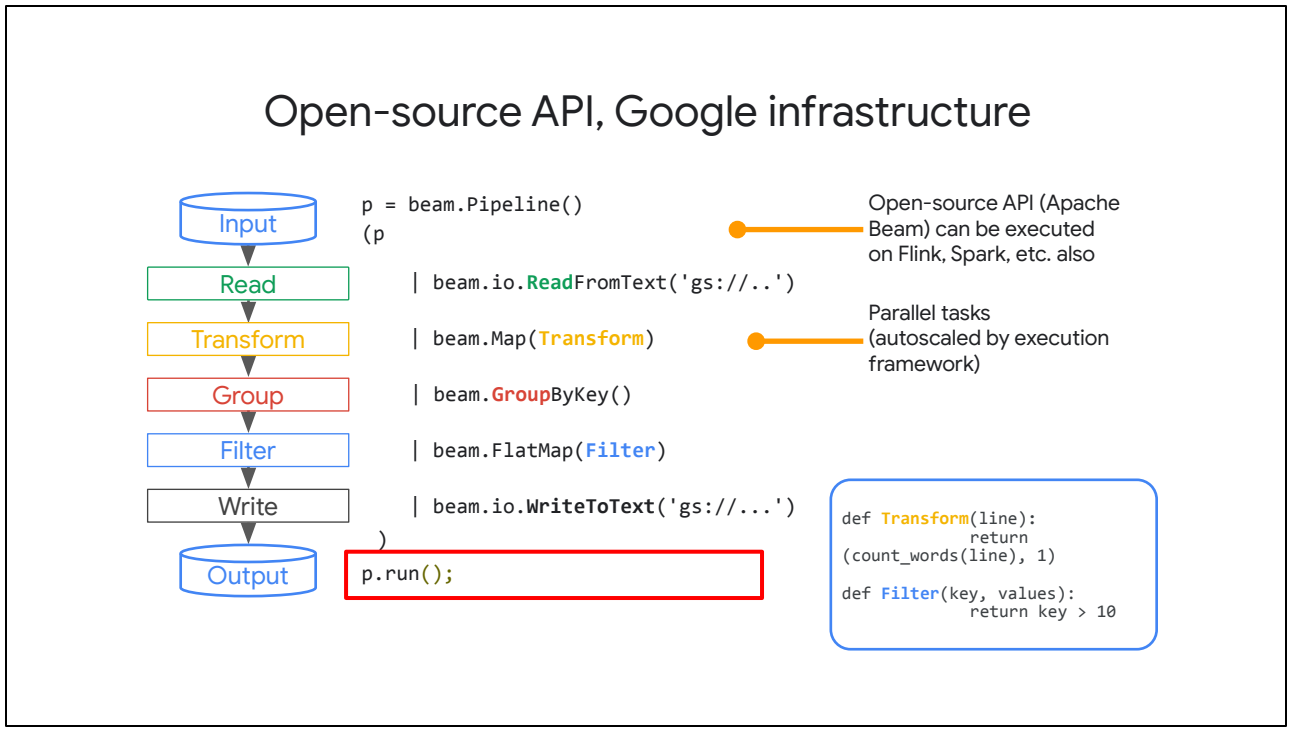


You can also filter out values, for example to ignore lines with fewer than 10 words.



Once all the transformation, grouping, and filtering operations are done, the pipeline writes the result to Google Cloud Storage.

Notice that this implementation separates the pipeline definition from the pipeline execution. All the steps that you see before call to the p.run() method are just defining what the pipeline should do.



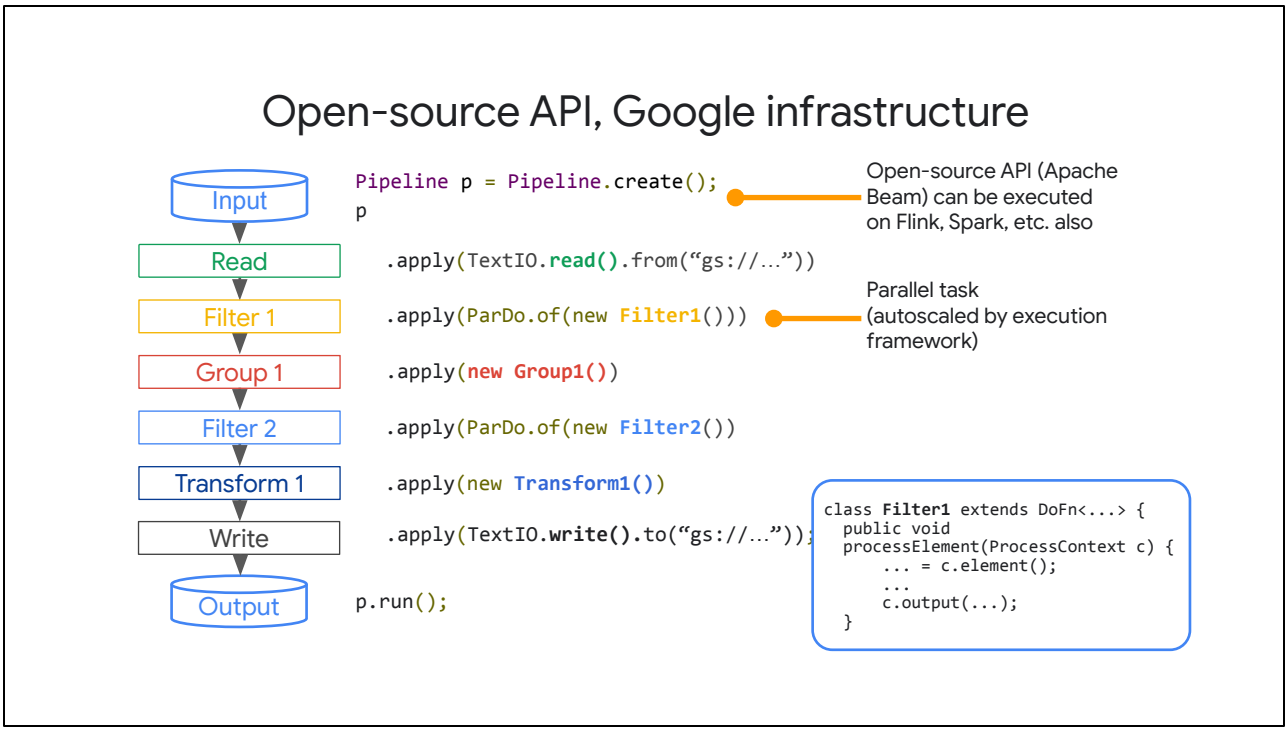
The pipeline actually gets executed only when you call the run method.

Notes:

Distinguish between the API (Apache Beam) and the implementation/execution framework (Dataflow)

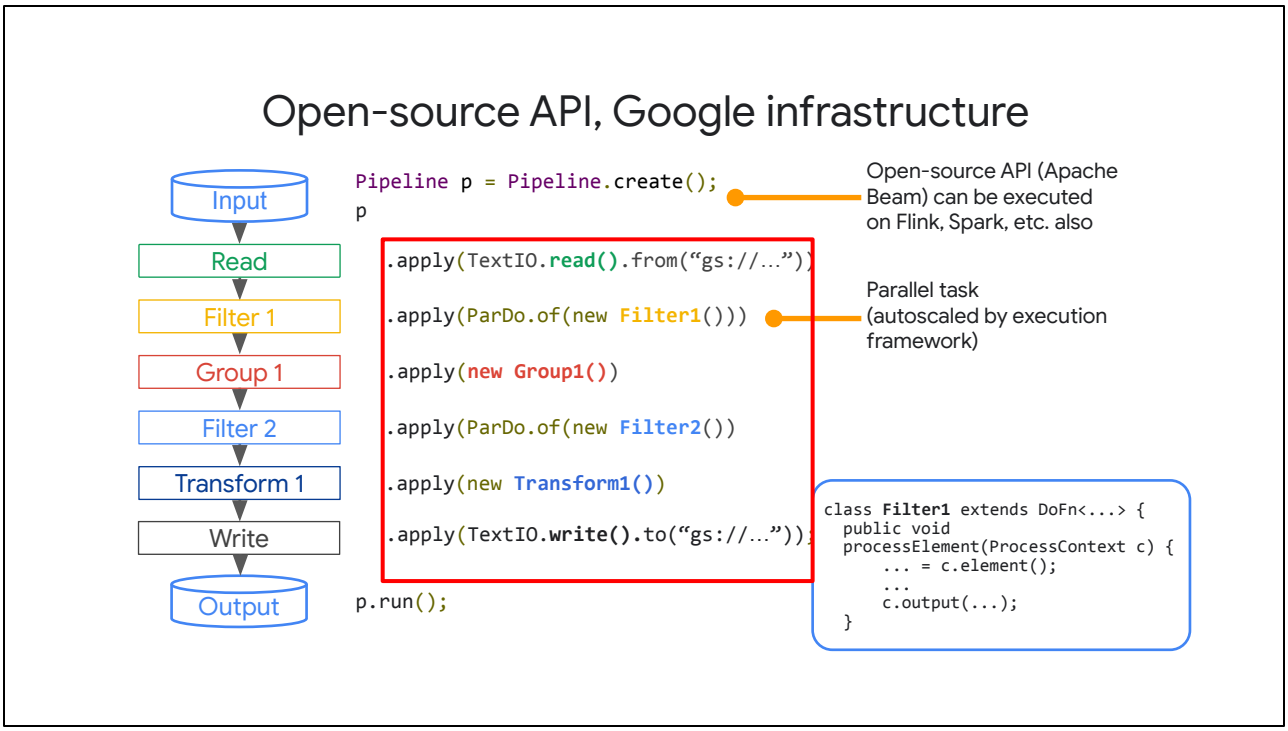
Each step of the pipeline does a filter, group, transform, compare, join, and so on. Transforms can be done in parallel.

c.element() gets the input. c.output() sends the output to the next step of the pipeline.



The way that you write code for Dataflow is using something called Apache Beam. Back in 2016, Google donated the Dataflow SDK, Software Development Kit to the Apache Software Foundation and helped launch an open source project called Apache Beam to help maintain and grow the APIs for programming data processing pipelines. So today, if you'd like to implement a data processing pipeline in Dataflow, you write your code using the Apache Beam API and then deploy the code to Cloud Dataflow.

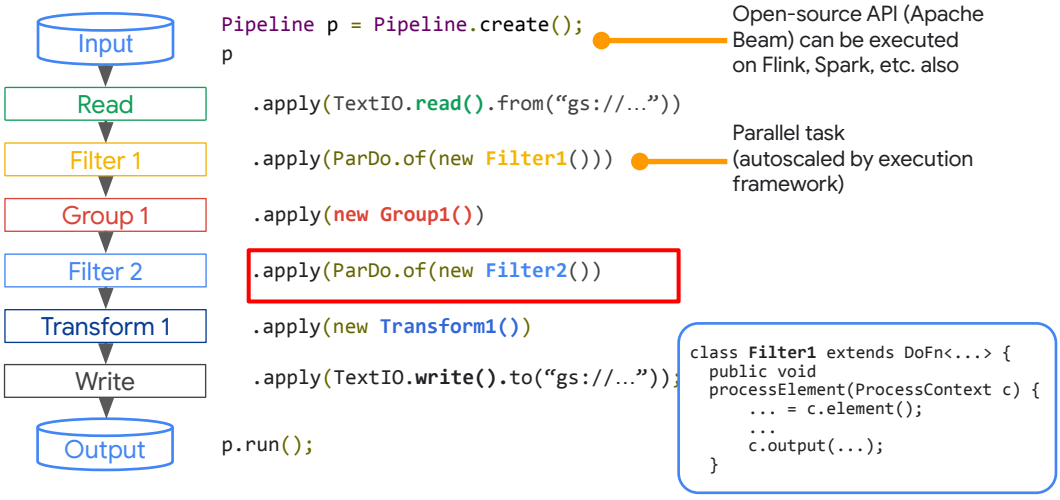
One thing that makes Apache Beam very easy to use is that conceptually it is very similar to how people think of data processing pipelines.



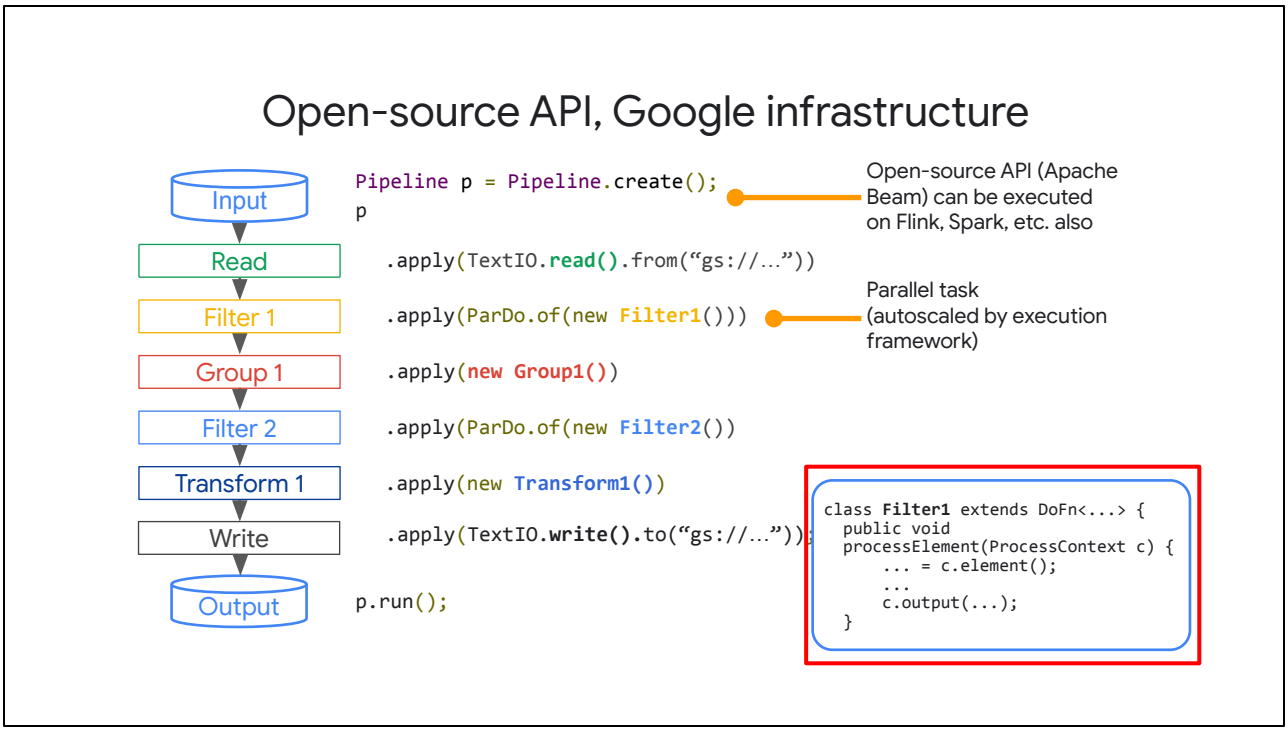
The way that you write code for Dataflow is using something called Apache Beam. Back in 2016, Google donated the Dataflow SDK, Software Development Kit to the Apache Software Foundation and helped launch an open source project called Apache Beam to help maintain and grow the APIs for programming data processing pipelines. So today, if you'd like to implement a data processing pipeline in Dataflow, you write your code using the Apache Beam API and then deploy the code to Cloud Dataflow.

One thing that makes Apache Beam very easy to use is that conceptually it is very similar to how people think of data processing pipelines. If you take a look at the source code in the center of the slide, this source code is implemented using Java and shows common steps for doing data transformations. For example, as an input you may want to read some data from an input like Google Cloud Storage and then filter out some of the values, maybe filter out rows of features that don't exist in your data set.

Open-source API, Google infrastructure

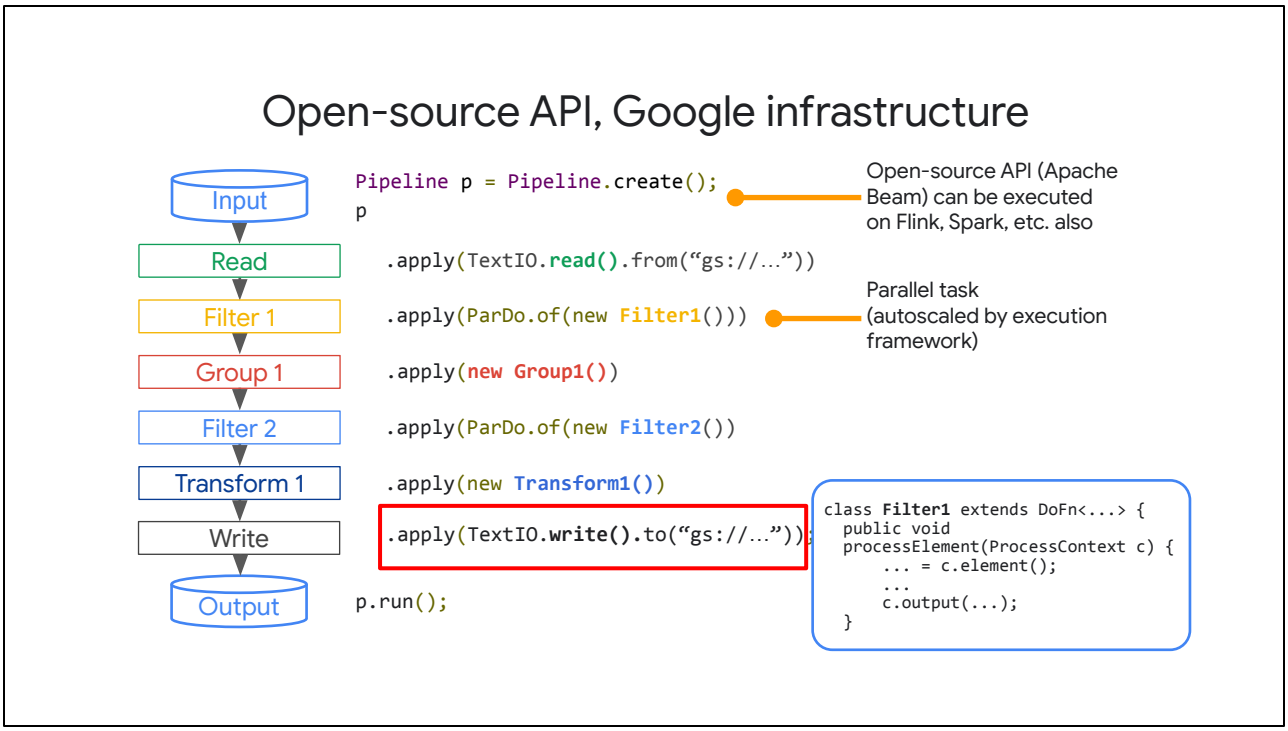


I'll come back and describe ParDo in more detail shortly.



Each one of the filters and other operations in Dataflow that can be scaled need to implement a DoFn interface. So in Java you simply extend the interface and in the body of the processElement function you pull in the element and generate the result. Once all the filtering, grouping, transformation operations have completed you write the result to Google Cloud Storage.

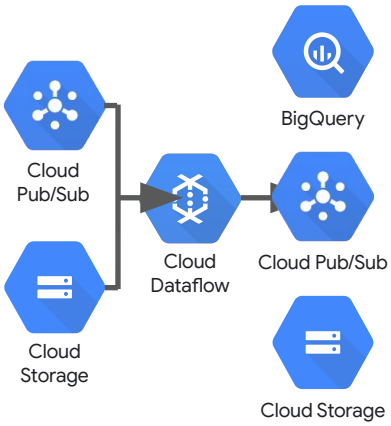
Notice that the read and write operations are done using something called a connector.



If you take a look at the TextIO class. TextIO can save raw text to outputs like Google Cloud Storage, or a file system. The Apache Beam provides a variety of connectors to help you use services on Google Cloud. Since Apache Beam is an open source project companies can implement their own connectors to other technologies

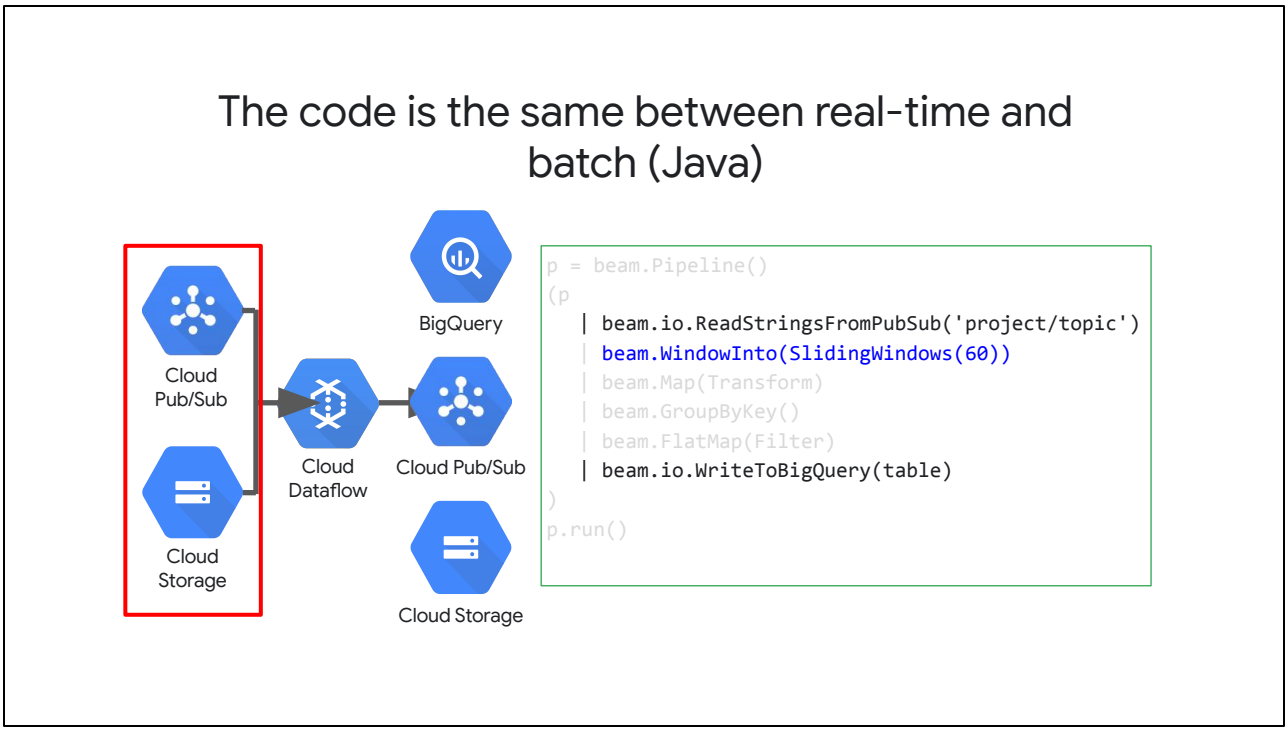
It is important to notice that this implementation separates the pipeline definition from the pipeline execution. All the steps that you see in the apply operations are just defining what the pipeline should do. Only when you call the run method does the pipeline actually get executed.

The code is the same between real-time and batch (Java)




```
p = beam.Pipeline()
(p
   | beam.io.ReadStringsFromPubSub('project/topic')
   | beam.WindowInto(SlidingWindows(60))
   | beam.Map(Transform)
   | beam.GroupByKey()
   | beam.FlatMap(Filter)
   | beam.io.WriteToBigQuery(table)
)
p.run()
```

One of the coolest things about Apache Beam is that it supports both



batch and streaming data processing using the same pipeline code. In fact, in the library's name, Beam comes from a contraction of Batch and Stream.

So why should you care? Well, it means that regardless of whether your data is coming from a batch data source like Google Cloud Storage or even from a streaming data source like PubSub, you can reuse the same pipeline logic. You can also output data to both batch and streaming data destinations. You can also easily change these data sources in the pipeline without having to change the logic of your pipeline implementation.

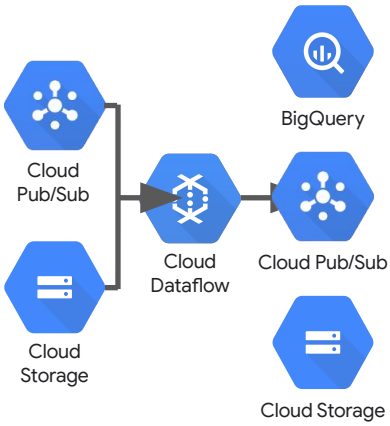
Here's how. Notice in the code on the screen that the read and write operations are done using beam.io methods. These methods use different connectors. For example, the PubSub connect can read the content of messages that are streamed into to the pipeline. Other connectors can read raw text from Google Cloud Storage, or a file system. The Apache Beam has a variety of connectors to help you use services on Google Cloud, like BigQuery. Also, since Apache Beam is an open source project, companies can implement their own connectors.
</KMO>

Notes:

You can get input from any of several sources, and you can write output to any of several sinks. The pipeline code remains the same.

You can put this code inside a servlet, deploy it to App Engine, and schedule a cron task queue in App Engine to execute the pipeline periodically.

The code is the same between real-time and batch (Java)




```
p = beam.Pipeline()
(p
   | beam.io.ReadStringsFromPubSub('project/topic')
   | beam.WindowInto(SlidingWindows(60))
   | beam.Map(Transform)
   | beam.GroupByKey()
   | beam.FlatMap(Filter)
   | beam.io.WriteToBigQuery(table)
)
p.run()
```

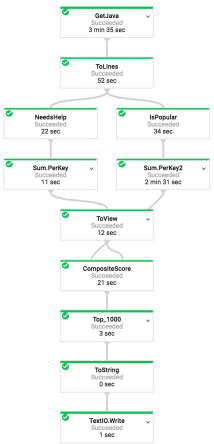
Here's how. Notice in the code on the screen that the read and write operations are done using beam.io methods. These methods use different connectors. For example, the PubSub connect can read the content of messages that are streamed into to the pipeline. Other connectors can read raw text from Google Cloud Storage, or a file system. The Apache Beam has a variety of connectors to help you use services on Google Cloud, like BigQuery. Also, since Apache Beam is an open source project, companies can implement their own connectors.

Notes:

You can get input from any of several sources, and you can write output to any of several sinks. The pipeline code remains the same.

You can put this code inside a servlet, deploy it to App Engine, and schedule a cron task queue in App Engine to execute the pipeline periodically.

Dataflow terms and concepts

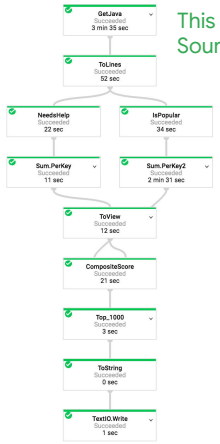


Before going too much further, let's cover some terminology that I will be using over and over throughout the rest of this module. You already know about the data processing pipelines which can run on Dataflow.

On the right hand side of the slide you can see the graphic for the pipeline.

Let's understand the pipelines in Apache Beam in more detail.

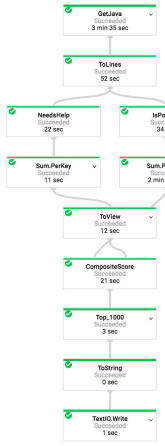
Dataflow terms and concepts



This is a Source

The pipeline must have a source which is where the pipeline gets the input data. The pipeline has a series of steps.

Dataflow terms and concepts

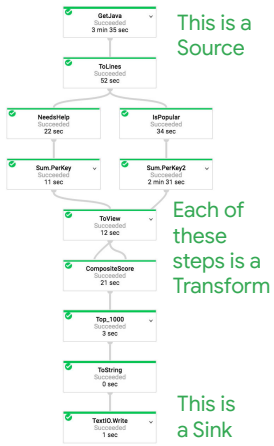


This is a Source

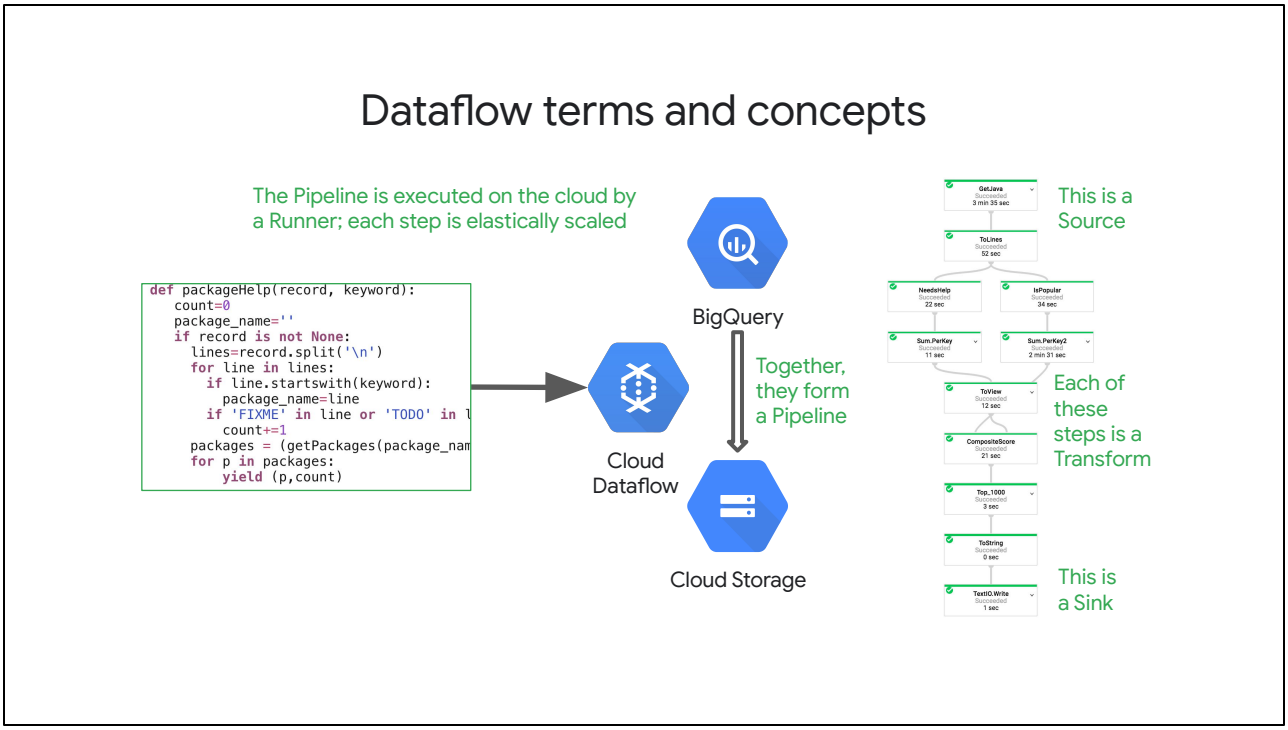
Each of these steps is a Transform

Each of the steps in Beam is called a transform.

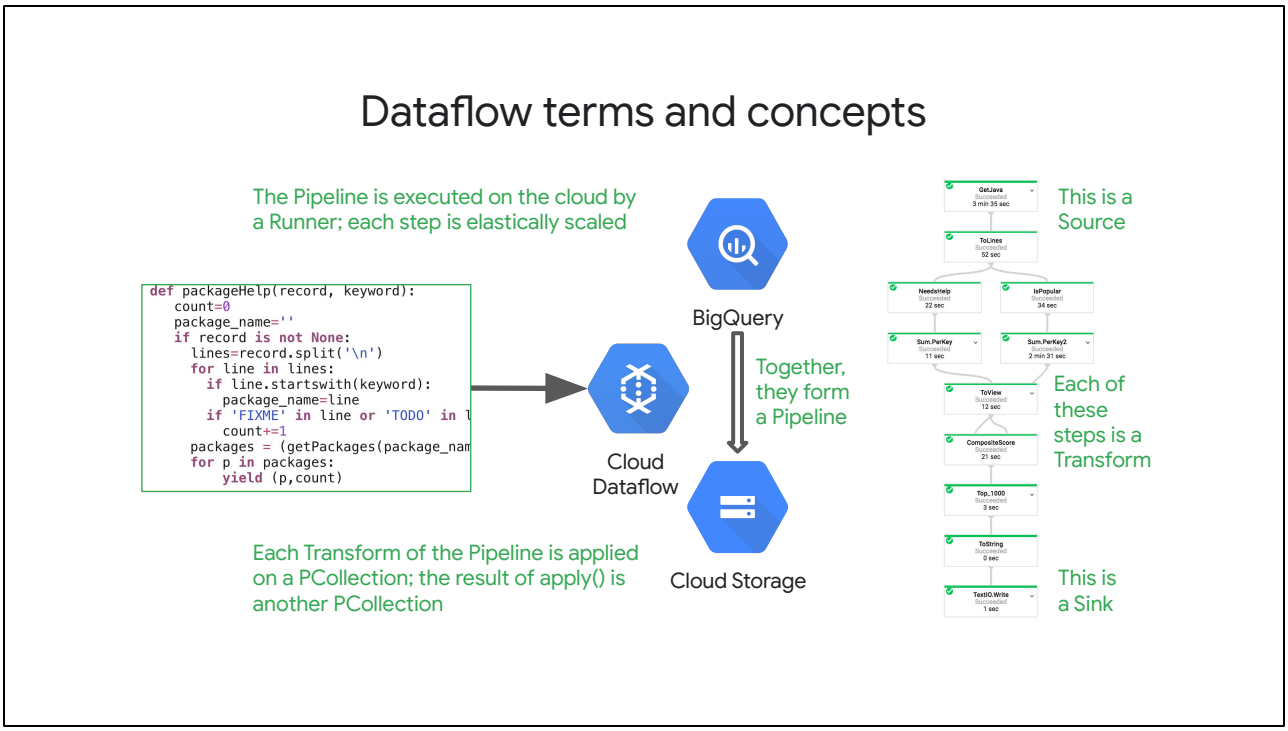
Dataflow terms and concepts



All of the transforms together generate some result which is written to something called a sink which is where the result is sent as an output of the pipeline.



To run a pipeline you need something called a runner. A runner takes the pipeline code and executes it. Runners is platform specific, meaning that there is a Dataflow runner for executing a pipeline on Cloud Dataflow, there is another runner if you want to use Apache Spark to run your pipeline. There is also a runner that will execute a pipeline on your local computer. If you'd like, you can even implement your own custom runner for your own distributed computing platform.



Each transform works on a data structure called a PCollection. I'll return to an explanation of a PCollection in detail shortly. For now just remember that every transform gets a PCollection as input and outputs the result to another PCollection.

Notes:

The idea is to write Java (or Python code), deploy it to Dataflow which then executes the pipeline.

The pipeline here reads data from BigQuery, does a bunch of processing and writes its output to CloudStorage.

Elastic: unlike Dataproc, there is no need to launch a cluster. Like BigQuery in that respect

Key concepts to be familiar with in Dataflow are highlighted in bold. Start at top-right and work your way clockwise through the callouts.

A Pipeline is a directed graph of steps

Read in data, transform it, write out

Can branch, merge, use if-then statements, etc.

Pythonic syntax

```
import apache_beam as beam
if __name__ == '__main__':

   # create a pipeline parameterized by commandline flags
   p = beam.Pipeline(argv=sys.argv)

   (p
      | 'Read' >> beam.io.ReadFromText('gs://...') # read input
      | 'CountWords' >> beam.FlatMap(lambda line: count_words(line))
      | 'Write' >> beam.io.WriteToText('gs://...') # write output
   )

   p.run() # run the pipeline
```

So how do you implement these pipelines? If you take a look at the code on the slide, you will notice that the first operation in the main method is beam.Pipeline which creates a pipeline instance. Once it is created, every transform is implemented as an argument to the apply method of the pipeline.

In the Python version of the Apache Beam library, the pipe operator is overloaded to call the apply method. That's why you have this funky syntax with pipe operators on top of each over. I like it. I think it is much easier to read this way. The strings like 'Read', 'CountWords', and 'Write' are just the human readable names that you can specify for each transform in the pipeline.

Notice that this pipeline is reading from and writing to Google Cloud Storage. And, as I pointed out earlier, none of the pipe operators actually run the pipeline. When you need your pipeline process some data, you need to call the the run method on the pipeline instance to execute it.
</KMO>

Notes:

| operator overloaded to mean .apply()
>> overload to mean "assign-this-name" to this PTransform is omitted here and introduced on next slide.

A Pipeline is a directed graph of steps

Read in data, transform it, write out

Can branch, merge, use if-then statements, etc.

```
import org.apache.beam.sdk.Pipeline;  // etc.

public static void main(String[] args) {
    // Create a pipeline parameterized by commandline flags.
    Pipeline p = Pipeline.create(PipelineOptionsFactory.fromArgs(args));

    p.apply(TextIO.read().from("gs://..."))   // Read input.
     .apply(new CountWords())                // Do some processing.
     .apply(TextIO.write().to("gs://..."));   // Write output.

    // Run the pipeline.
    p.run();
}
```

<KMO>So how do you implement these pipelines. If you take a look at the pipeline implementation on the slide, you will notice that you first need to create a pipeline. It is very easy to create a pipeline. you call pipeline create and pass in some configuration options. The options can be created from the command line arguments. Once the pipeline is created, every transform in the pipeline is implemented as an argument to the apply method. So notice here in this pipeline, the source of the pipeline is supported with the TextIO connector, reading from Google Cloud Storage. Then once the input is read, the next transform simply counts the words in the input text. The final step of the pipeline, the sink, does output to google cloud storage. As I pointed out earlier it is important to recognize that all the calls to the apply don't actually run the pipeline. If you need to have the pipeline to actually process some data, you need to call the run method on the pipeline and that will actually execute it. </KMO>

Python API conceptually similar

Read in data, transform it, write out

Pythonic syntax

```
import apache_beam as beam

if __name__ == '__main__':
   # create a pipeline parameterized by commandline flags
   p = beam.Pipeline(argv=sys.argv)

   (p
      | beam.io.ReadFromText('gs://...') # read input
      | beam.FlatMap(lambda line: count_words(line)) # do some processing
      | beam.io.WriteToText('gs://...') # write output
   )

   p.run() # run the pipeline
```

<KMO>The implementation in Python is very similar. You create the pipeline using the beam.Pipeline helped method. In Python, less verbose and using more of Pythonic style. Another thing to notice is that instead of using the apply method, in Python the pipe operator is overloaded to call apply that's why you have this funky syntax with pipes on top of each over. I like it. I think it is much easier to read this way.

Notes:

| operator overloaded to mean .apply()
>> overload to mean “assign-this-name” to this PTransform is omitted here and introduced on next slide.

Apply Transform to PCollection

Data in a pipeline are represented by PCollection

- Supports parallel processing
- Not an in-memory collection; can be unbounded

```
lines = p | ...
```

Apply Transform to PCollection; returns PCollection

```
sizes = lines | 'Length' >> beam.Map(lambda line: len(line) )
```

As I mentioned earlier, every time you use the pipe operator, you provide a PCollection data structure as input and return a PCollection as output. An important thing to know about PCollections is that unlike many data structures, PCollection does not store all of its data in memory. Remember that Dataflow is elastic and can use a cluster of servers for your pipeline, so PCollection is like a data structure with pointers to where in the Dataflow cluster your data is stored. That's how Dataflow can provide elastic scaling of the pipeline.

Let's say we have a PCollection of lines. For example, the lines could come from file in Google Cloud Storage. One way to implement a transformation is to take a PCollection of strings which are called lines in the code and return a PCollection of integers.

This specific transform step in the code computes the length of each line.
</KMO>

Notes:

Java on previous slide; Python on this slide.

In Python, do not use apply(). Best practice is to use the pipe operator.

Notice how we supply the name ‘Length’ to the bottom transform.

Apply Transform to PCollection

Data in a pipeline are represented by PCollection

- Supports parallel processing
- Not an in-memory collection; can be unbounded

```
PCollection<String>  lines = p.apply(...)   //
```

I mentioned earlier that every transform takes a PCollection as input and returns a PCollection as output. An important thing to know about PCollections is that unlike many data structures, PCollection does not store all of its data in memory. Remember that Dataflow is elastic and can use a cluster of servers for your pipeline, so PCollection is like a data structure with pointers to where in the Dataflow cluster your data is stored. That's how Dataflow can provide elastic scaling of the pipeline.

Let's say we have a PColllection of lines. For example, here the lines are read from a file in Google Cloud Storage. One way to implement a transformation is to in Java is to take strings as line, strings of line and return an integer.

Apply Transform to PCollection

Apply Transform to PCollection; returns PCollection

```
PCollection<Integer> sizes =
                  lines.apply("Length", ParDo.of(new DoFn<String,
Integer>() {
                                      @ProcessElement
                                      public void
processElement(ProcessContext c) throws Exception {
                                                                    S
tring line = c.element();
                                                                    c
.output(line.length());
}}))
```

This specific transform will compute the length of each line. Notice that in order to get access to the strings of the lines themselves inside of the process Element method, you call c.element. To pass the result to the next tranform in the pipeline, you need to class c.output and pass in the resulting value.

The important thing to remember here is that there is no return statement, we are simply working with the PCollection. Apache Beam will wrap the output into another PCollection. Another important thing to notice here is that the implementation of the DoFn interface is passed to the ParDo.of. Is wrapped with the static method ParDo.of. ParDo is short for Parallel Do. This is a hint that you can provide as a developer to Dataflow that the code can be elastically scaled and distributed across a cluster of servers.
</KMO>

Apply Transform to PCollection (Python)

Data in a pipeline are represented by PCollection

- Supports parallel processing
- Not an in-memory collection; can be unbounded

```
lines = p | ...
```

Apply Transform to PCollection; returns PCollection

```
sizes = lines | 'Length' >> beam.Map(lambda line: len(line) )
```

Python version of the code is conceptually the same but less verbose. Notice that it is using the Python pipe operator instead of the apply method.

The implementation takes the incoming argument and returns back the length of the line.

Notes:

Java on previous slide; Python on this slide.

In Python, do not use apply(). Best practice is to use the pipe operator.

Notice how we supply the name ‘Length’ to the bottom transform.

Ingesting data into a pipeline (Python)

Read data from file system, GCS or BigQuery

Text formats return String

```
lines = beam.io.ReadFromText('gs://.../input-*.csv.gz')
```

BigQuery returns a TableRow

```
rows = beam.io.Read(beam.io.BigQuerySource(query='SELECT x, y, z ' \
                'FROM [project:dataset.tablename]', project='PROJECT'))
```

As you already know, Apache Beam SDK comes with a variety of connectors that enable Dataflow to read from a many of data sources, including text files in Google Cloud Storage or file systems. With different connectors it is possible to read even from real time streaming data sources like Google Cloud PubSub or Kafka.

One of the connectors is for BigQuery, the data warehouse on GCP. When using the BigQuery connector, you need to specify the SQL statement that BigQuery will evaluate to return back a table with rows of results. The TableRows are then passed to the pipeline in a PCollection.

Notes:

Notice the wildcards and .gz extension -- both are supported.

There is also I/O to Bigtable, but it's not part of the Dataflow SDK

Ingesting data into a pipeline (Java)

Read data from file system, GCS, BigQuery, Pub/Sub

Text formats return String

```
PCollection<String> lines = p.apply(TextIO.read().from("gs://.../input-*.csv.gz");
```

```
PCollection<String> lines = p.apply(PubsubIO.readStrings().fromTopic(topic));
```

BigQuery returns a TableRow

```
String javaQuery = "SELECT x, y, z FROM [project:dataset.tablename]";
PCollection<TableRow> javaContent = p.apply(BigQueryIO.read().fromQuery(javaQuery))
```

As I mentioned earlier, the Apache Beam SDK provides a variety of connectors that enable Dataflow to read from a variety of data sources, including text files in Google Cloud Storage or file systems. With different connectors it is possible to read even from real time streaming data sources like Google Cloud PubSub or Kafka. There is a connector to read from BigQuery, which is a data warehouse on GCP.

When using the BigQuery connector, you need to specify the SQL statement that BigQuery will evaluate to return back a table with rows of results. The TableRows are then passed to the pipeline in a PCollection.

Notes:

Notice the wildcards and .gz extension -- both are supported.

There is also I/O to Bigtable, but it's not part of the Dataflow SDK

Can write data out to same formats (Python)

Write data to file system, GCS or BigQuery

```
beam.io.WriteToText(file_path_prefix='/data/output', file_name_suffix='.txt')
```

Can prevent sharding of output (do only if it is small)

```
beam.io.WriteToText(file_path_prefix='/data/output',
file_name_suffix='.txt', num_shards = 1)
```

The output must be a PCollection of Strings before writing out

To export out the result of a pipeline, there are connectors for Cloud Storage, PubSub, BigQuery and more. Of course you can also just write the result to a file system.

An important thing to keep in mind when writing to a file system, is that Dataflow can distribute execution of your pipeline across a cluster of servers. This mean that there can be multiple servers trying to write results to the file system. In order to avoid contention issues where multiple servers are trying to get a file lock to the same file concurrently, by default the TextIO connector will shard the output, writing the results across multiple files on the file system. For example, here the pipeline is writing the result to a file with a prefix output in the data director. Let's say there is a total of 10 file will be written, so Dataflow will write files like output0of10.txt, output1of10.txt and so on.

withoutSharding method as hint

Keep in mind that if you do that, you will have the file lock contention issue that I mentioned earlier, so it only makes sense to use the withoutSharding writes when working with smaller data sets that can be processed on a single node.
</KMO>

Notes:

Normally, you'll get /data/output-0000-of-0010.txt

Can write data out to same formats (Java)

Write data to file system, GCS, BigQuery, Pub/Sub

```
lines.apply(TextIO.write().to("/data/output").withSuffix(".txt"))
```

Can prevent sharding of output (do only if it is small)

```
.apply(TextIO.write().to("/data/output").withSuffix(".csv").withoutSharding())
```

May have to transform PCollection<Integer>, etc. to PCollection<String> before writing out

There is also a variety of connectors for writing out results of a pipeline. For example, in addition to writing to Cloud Storage, PubSub, or BigQuery, you can also write the result to a file system.

An important thing to keep in mind when writing to a file system, is that Dataflow can distribute execution of your pipeline across a cluster of servers. This mean that there can be multiple servers trying to write results to the file system. In order to avoid contention issues where multiple servers are trying to get a file lock to the same file concurrently, by default the TextIO connector will shard the output, writing the results across multiple files on the file system. For example, here the pipeline is writing the result to a file with a prefix output in the data director. Let's say there is a total of 10 file will be written, so Dataflow will write files like output0of10.txt, output1of10.txt and so on.

withoutSharding method as hint

Keep in mind that if you do that, you will have the file lock contention issue that I mentioned earlier, so it only makes sense to use the withoutSharding writes when working with smaller data sets that can be processed on a single node.
</KMO>

Notes:

Normally, you'll get /data/output-0000-of-0010.txt

Executing pipeline (Java)

Simply running main() runs pipeline locally

```
java -classpath ...        com...
```

```
mvn compile -e exec:java -Dexec.mainClass=$MAIN
```

To run on cloud, submit job to Dataflow

```
mvn compile -e exec:java \
      -Dexec.mainClass=$MAIN \
      -Dexec.args="--project=$PROJECT \
      --stagingLocation=gs://$BUCKET/staging/ \
      --tempLocation=gs://$BUCKET/staging/ \
      --runner=DataflowRunner"
```

Executing the pipeline is very straight forward. In Java you have two options for running the pipeline. One way is to manually try to setup up the classpath to include all the dependencies for Beam and other code needed to run your main method. This can get complex very quickly. The second option is to use a pre-build pom.xml which can be used together with Maven to compile and run your pipeline code. This way you don't have to worry about getting your dependencies and configuring the classpath. As long as you are using Maven you will be able to compile and run your pipeline locally using the mvn exec:java command.

To submit the pipeline as a job to execute in dataflow, you need to provide some additional information. You need to include arguments with the name of the project, and location in a google cloud storage bucket where dataflow will keep some staging and temporary data. You also need to specify the name of the runner which in this case is the DataflowRunner.
</KMO>

Notes:

Run using java and specifying classpath etc. or use mvn

Specify project for billing and staging, temporary locations to store intermediate output, and runner as Dataflow.

Executing pipeline (Python)

Simply running main() runs pipeline locally

```
python ./grep.py
```

To run on cloud, specify cloud parameters, and submit the job to Dataflow

```
python ./grep.py \
      --project=$PROJECT \
      --job_name=myjob \
      --staging_location=gs://$BUCKET/staging/ \
      --temp_location=gs://$BUCKET/staging/ \
      --runner=DataflowRunner
```

With a pipeline implemented in Python, you can run the code directly in the shell using the python command.

To submit the pipeline as a job to execute in Dataflow on GCP, you need to provide some additional information. You need to include arguments with the name of the GCP project, location in a google cloud storage bucket where dataflow will keep some staging and temporary data. And you also need to specify the name of the runner which in this case is the DataflowRunner.
</KMO>

Notes:

Conceptually similar to Java.

Syntax is pythonic: --staging_location instead of --stagingLocation etc.

Executing pipeline (Python)

Simply running main() runs pipeline locally

```
python ./grep.py
```

To run on cloud, specify cloud parameters

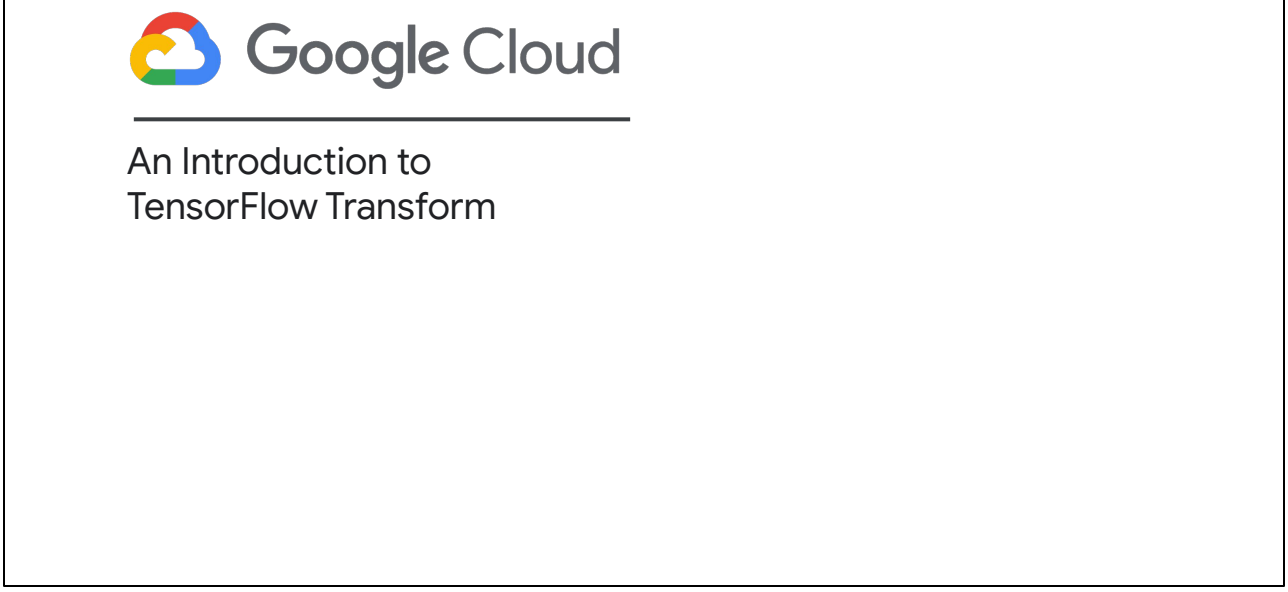
```
python ./grep.py \
      --project=$PROJECT \
      --job_name=myjob \
      --staging_location=gs://$BUCKET/staging/ \
      --temp_location=gs://$BUCKET/staging/ \
      --runner=DataflowRunner
```

In case of Python, you can just run the code directly using the python command. To run on Dataflow you just pass in the same arguments as in case of Java

Notes:

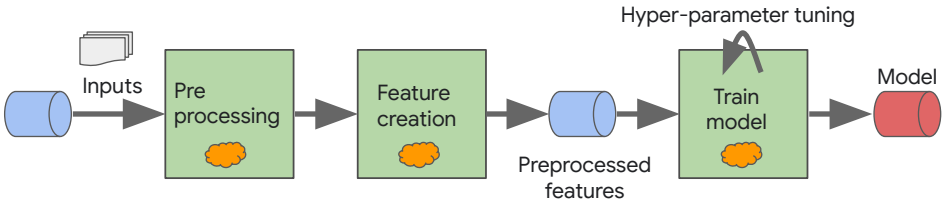
Conceptually similar to Java.

Syntax is pythonic: --staging_location instead of --stagingLocation etc.

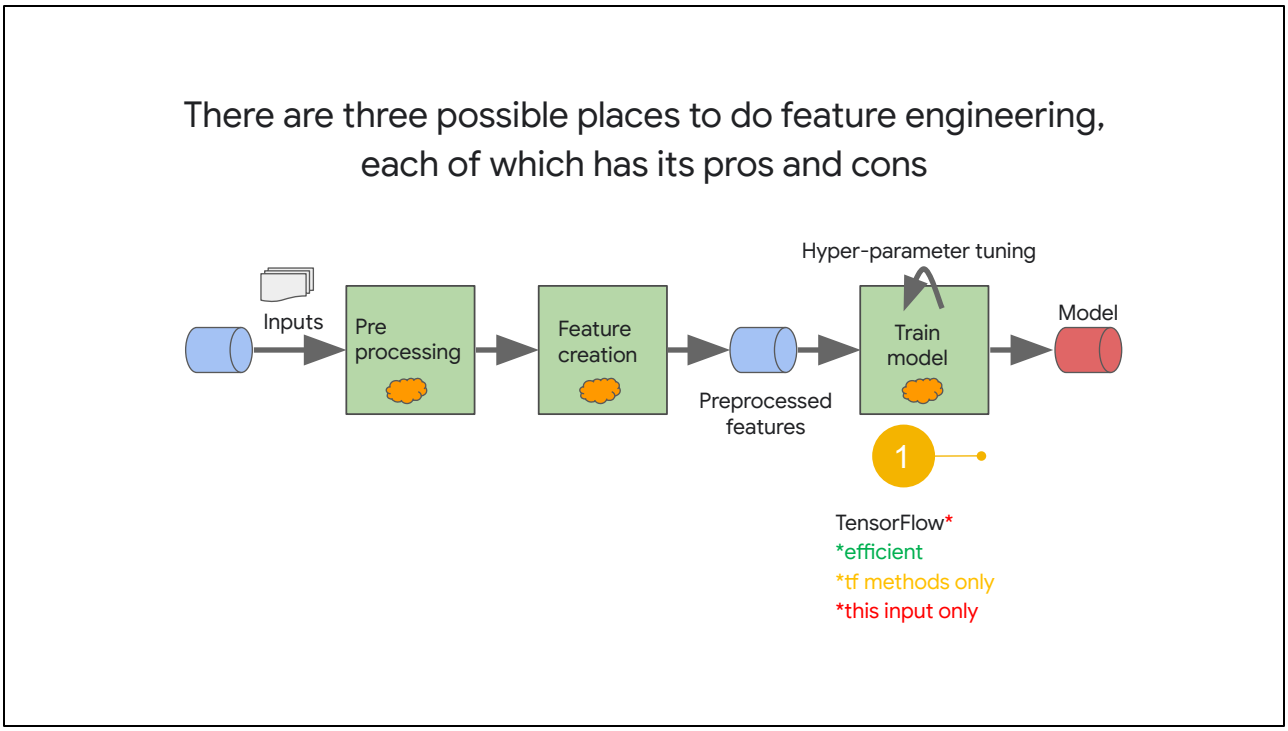


Welcome back! I'm Lak and we were talking about three different ways to implement Feature Enginering.

There are three possible places to do feature engineering, each of which has its pros and cons



Recall that we were talking about three possible places to do feature engineering.
We said that you could do feature engineering within TensorFlow itself



using feature columns or by wrapping the feature dictionary and adding arbitrary TensorFlow code. This is great, because it is efficient -- TensorFlow code on a GPU or TPU!

But why do I say arbitrary *TensorFlow* code?

Because this needs to be code that is executed as part of model function, as part of the TensorFlow graph.

So, you can not do a query on your corporate database and stick a value in there.

Well you could write a custom TensorFlow op in C++ and call it, but … let's ignore that for now.

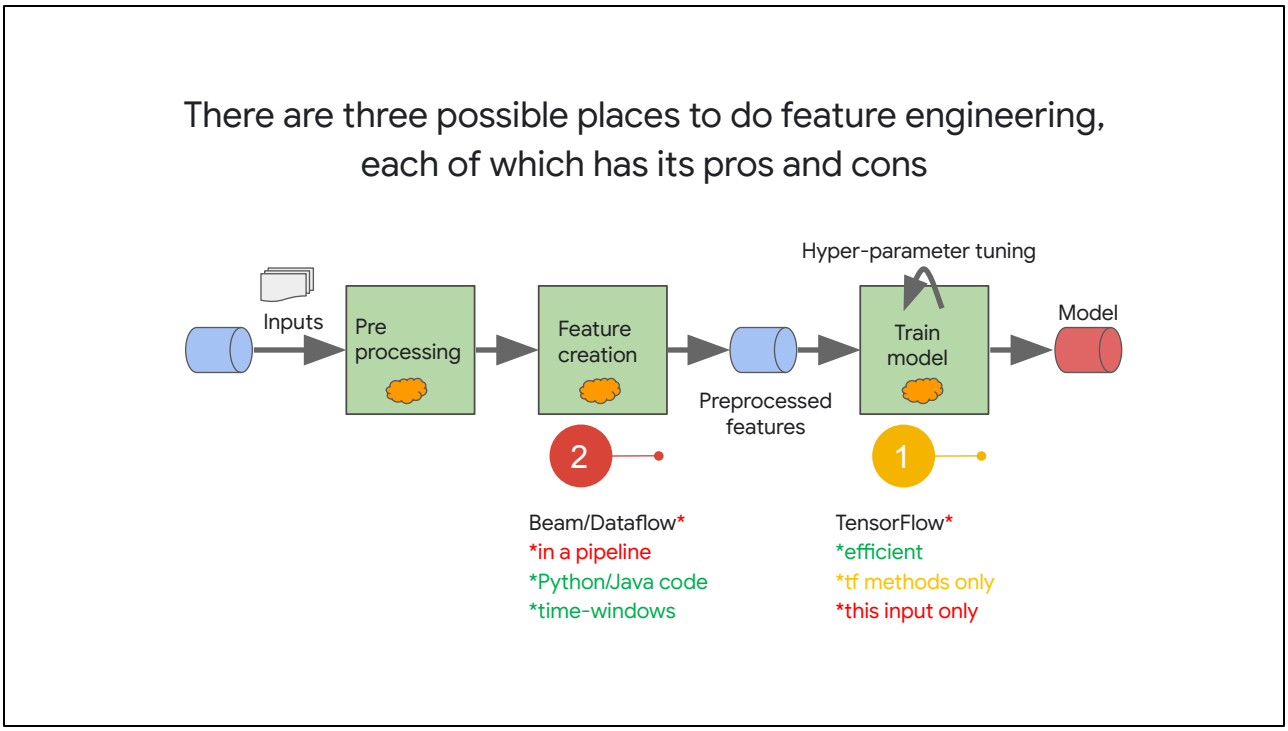
Also, you can only do things that rely on this input value and this input value alone.

So, if you want to compute a rolling average? Well that's kinda hard to do.

Later, we'll look at sequence models where it appears we are processing a time series, so multiple input values, btu the input there is the entire sequence.

So, the limit here is that we can do preprocessing on a single input only.

TensorFlow models (sequence models are an exception), TensorFlow models tend to be stateless.

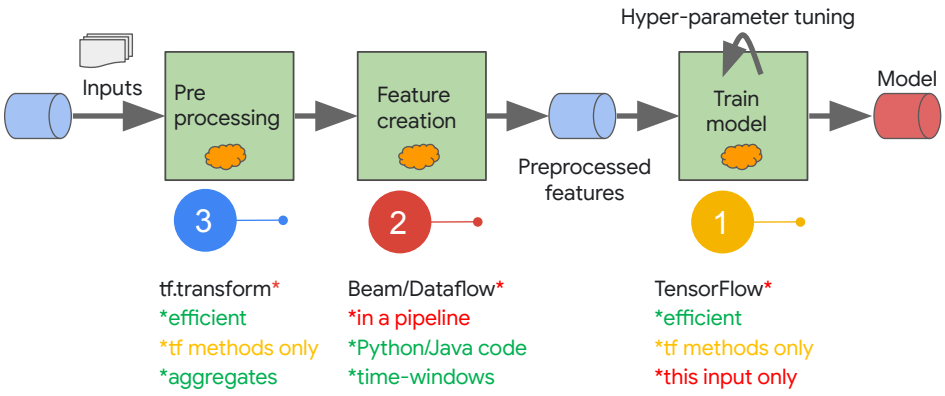


In past two chapters, we also looked at how to do preprocessing or feature creation in Apache Beam on Cloud Dataflow.

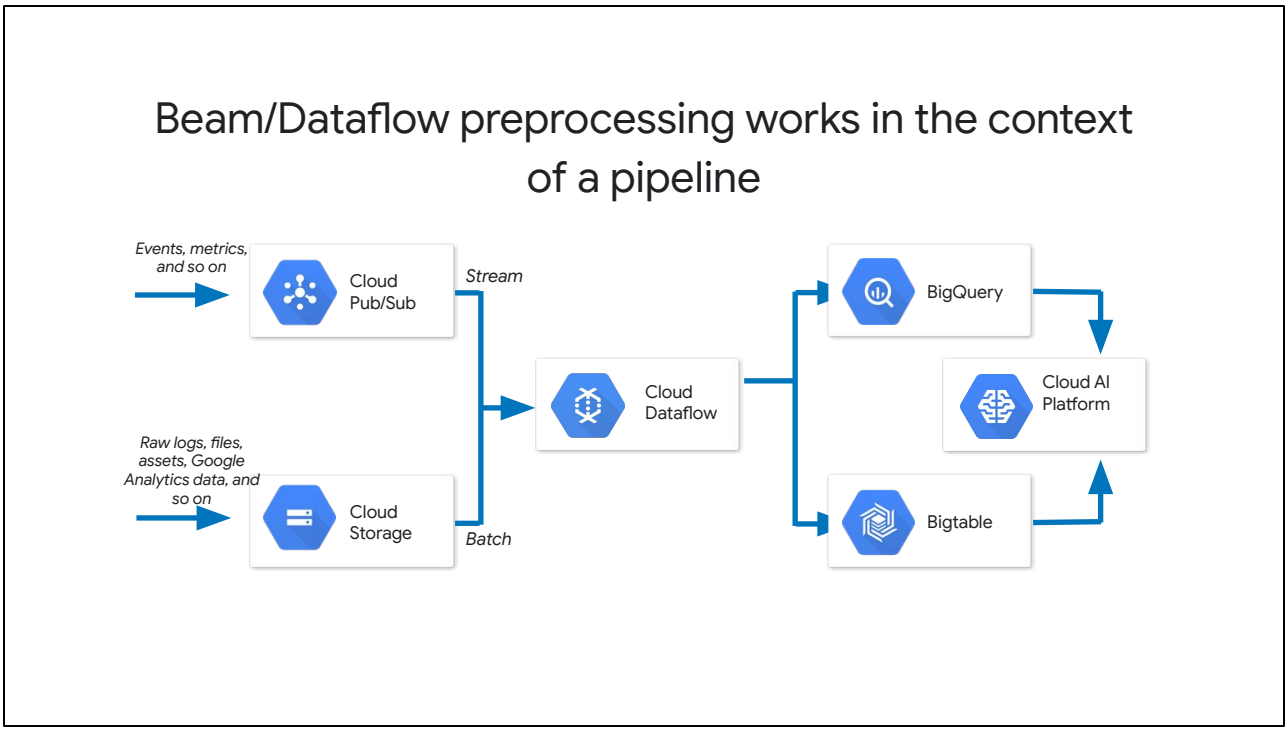
Dataflow lets us execute arbitrary Python or Java code and allows us to handle multiple input values in a stateful way. For example, you can compute a time-windowed average, like the average number of bicycles at a traffic intersection over the past hour. However, you will have to run your prediction code also within a pipeline. This is good for examples like time-windowed averages, where you need a pipeline in any case.

But what if all you want is the min/max to scale the values or the vocabulary to convert categorical values into numbers? Running a Dataflow pipeline in prediction seems a bit like overkill.

There are three possible places to do feature engineering, each of which has its pros and cons



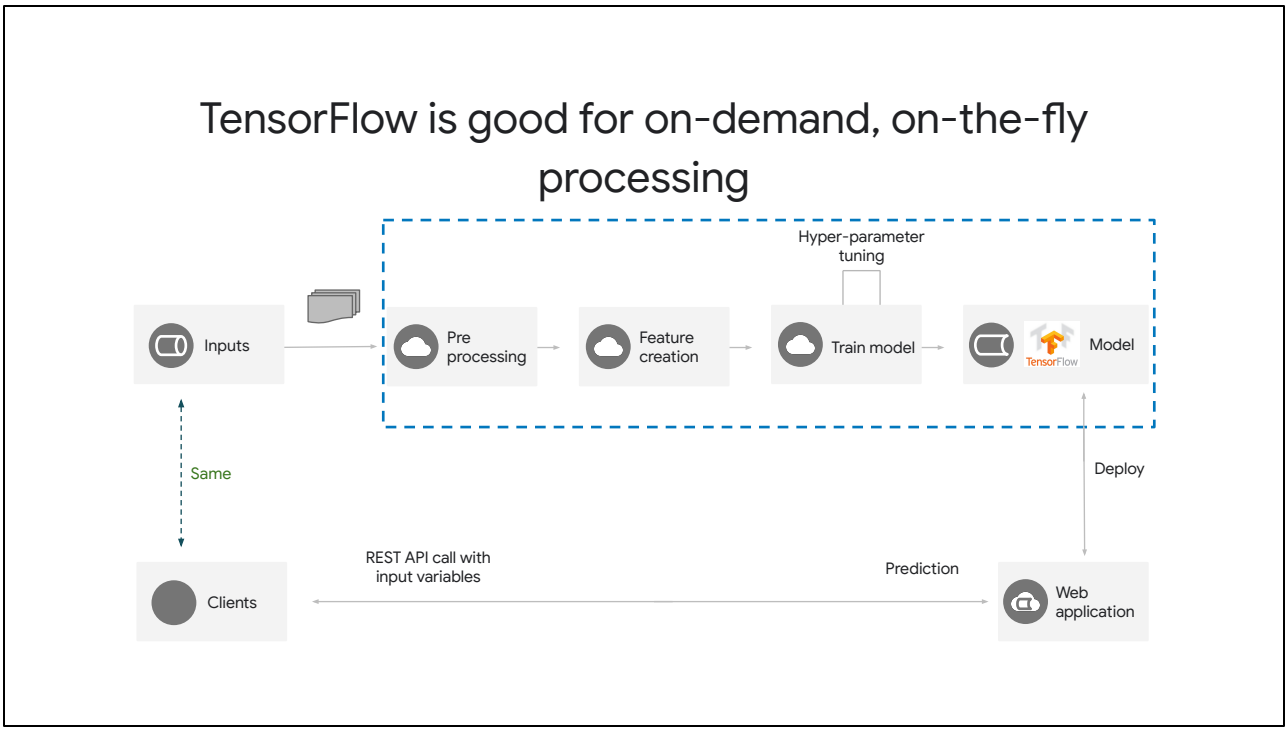
Enter tf.transform which is a hybrid of the first two approaches.
With TensorFlow Transform, you are limited to tf methods, but then you get the efficiency of TensorFlow. You can also use the aggregate of your entire training dataset because tf.transform uses Dataflow during training, but only TensorFlow during prediction.
Let's look at how TensorFlow Transform works.



TensorFlow transform is a hybrid of Apache Beam and TensorFlow. It's in between the two.

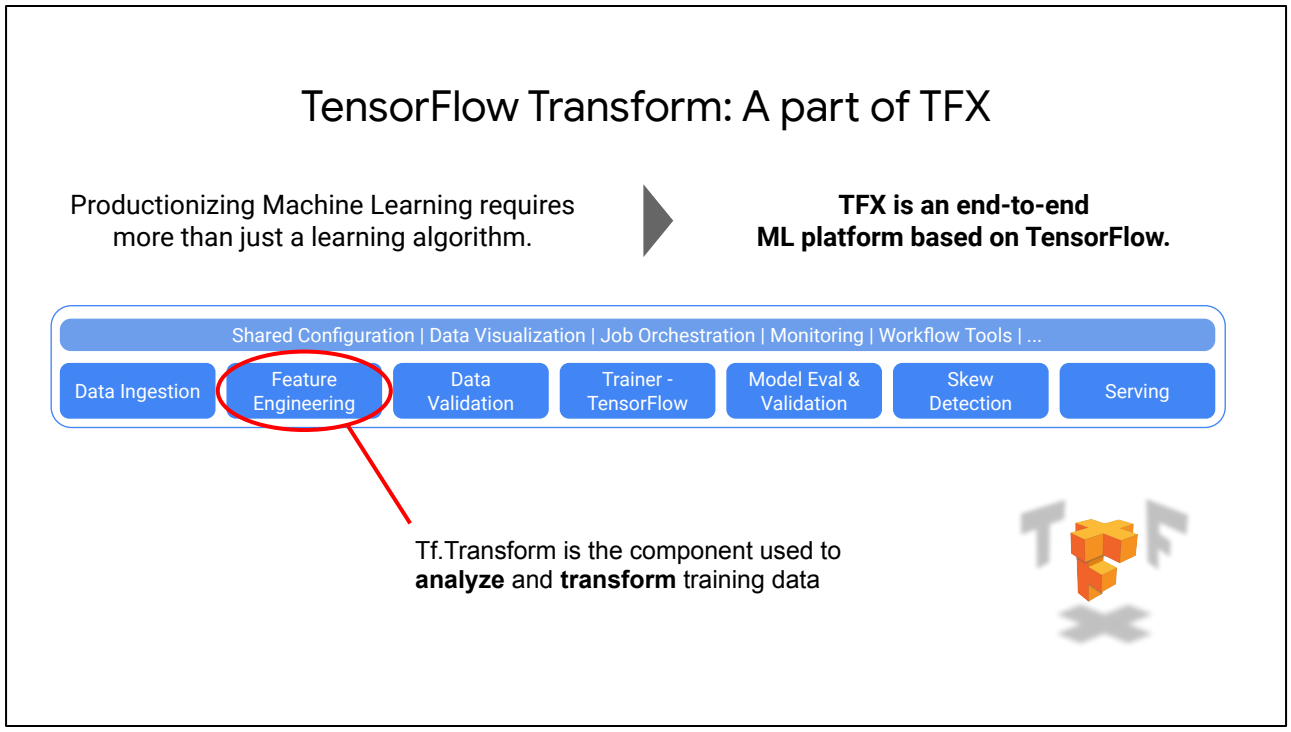
Dataflow preprocessing only works in the context of a pipeline. Think in terms of incoming, streaming data such as IoT data or flights data. The Dataflow pipeline might invoke the predictions and save the predictions to Bigtable; these predictions are then served to anyone who visits the webpage in the next 60 seconds, at which point a new prediction is available in Bigtable.

In other words, think of backend preprocessing for ML models in Dataflow. You use Dataflow for preprocessing that needs to maintain state (such as time-windows)



Think on-the-fly preprocessing for ML models in TensorFlow. You use TensorFlow for preprocessing that is based on the provided input only.

If you put all the stuff in the dotted box into the TensorFlow graph, then it is quite easy for clients to just invoke a web application and get all the processing handled for them.

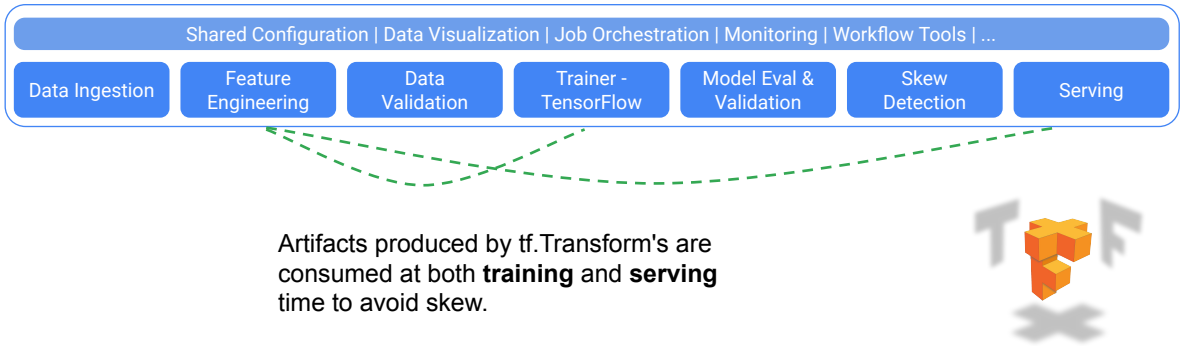


Tf.Transform is the component used to **analyze** and **transform** training data.

TensorFlow Transform: A part of TFX

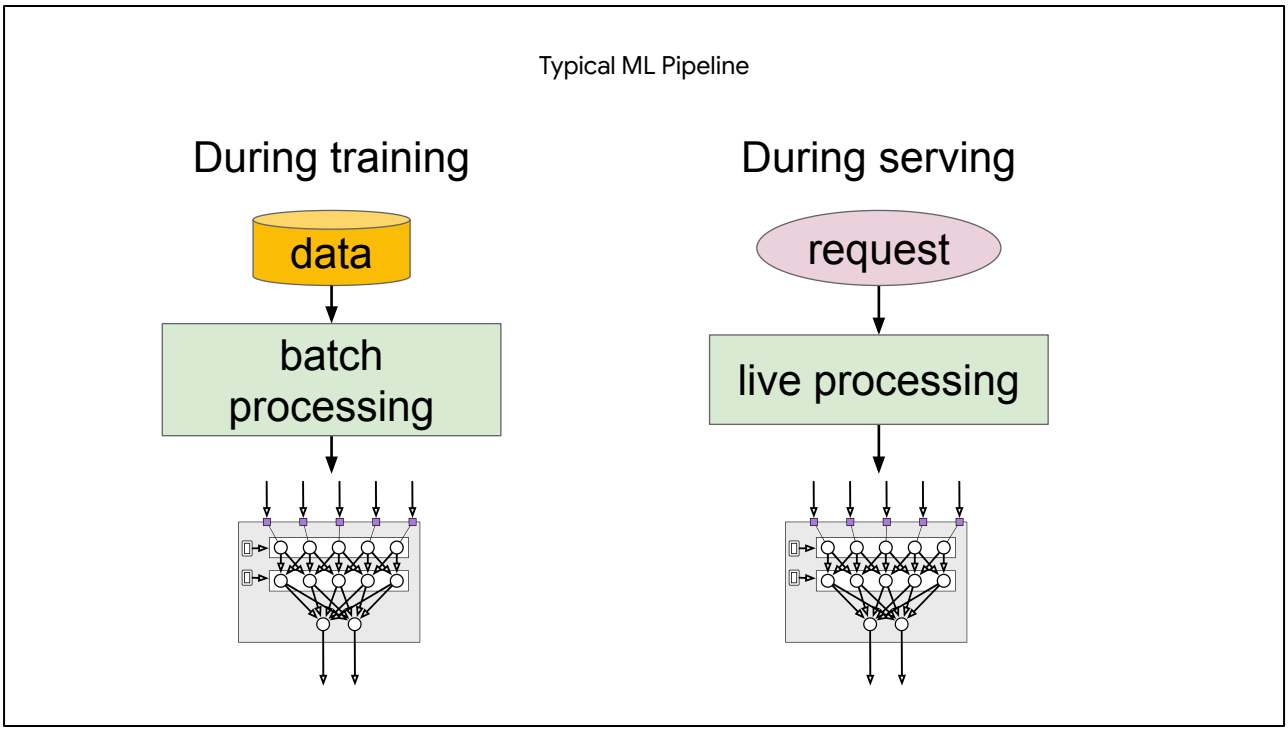
Productionizing Machine Learning requires more than just a learning algorithm.

TFX is an end-to-end ML platform based on TensorFlow.



Artifacts produced by tf.Transform's are consumed at both **training** and **serving** time to avoid skew.

Artifacts produced by tf.Transform's are consumed at both **training** and **serving** time to avoid skew.



Here is a typical ML Pipeline.

Problems

- Need to keep batch and live processing in sync.
- All other tooling (e.g. evaluation) must also be kept in sync with batch processing.

Problems with a typical ML pipeline:

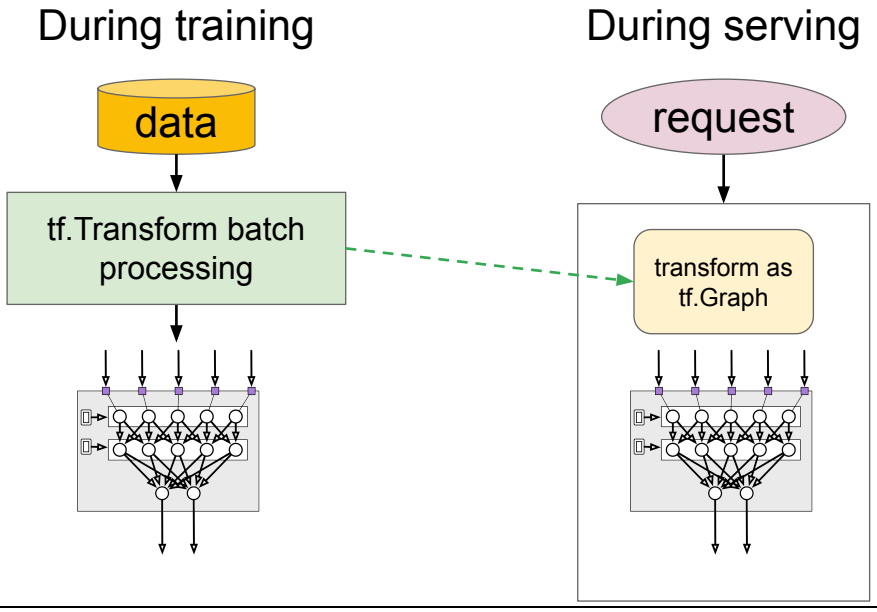
- Need to keep batch and live processing in sync.
- All other tooling (e.g. evaluation) must also be kept in sync with batch processing.

Partial Solutions

- Do everything in the training graph
- Do everything in the training graph + using statistics/vocabs generated from raw data.

Problems with “do everything in the training graph”: (1) this loses the benefits of materialization, (2) doesn’t allow for “reduces” (more on this in beam vs TF).
Problems with “do everything in the training graph + using statistics/vocabs generated from raw data”: (1) only allows for stats/vocabs on raw data (2) still doesn’t address materialization.

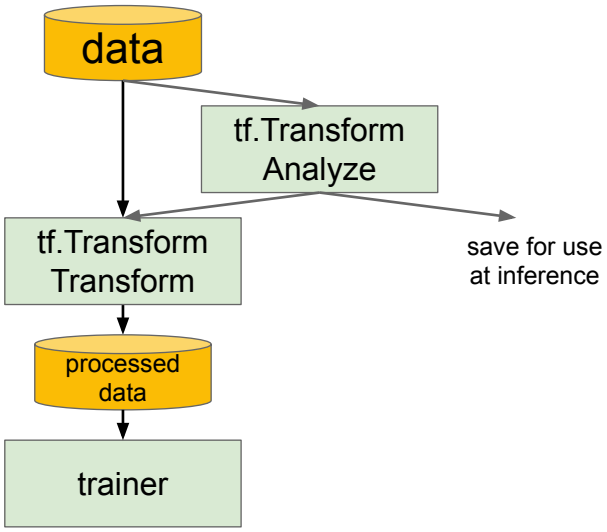
tf.Transform



Transform does batch processing, but also emits a TF graph that can be used to “repeat” these transformation in serving. More on what “repeat” means later.

By combining this graph with the trained model graph, in to a single serving graph, you can guarantee that the same operations that were done to the training data, will be done to the request during serving, before passing the transformed result to the trained model graph.

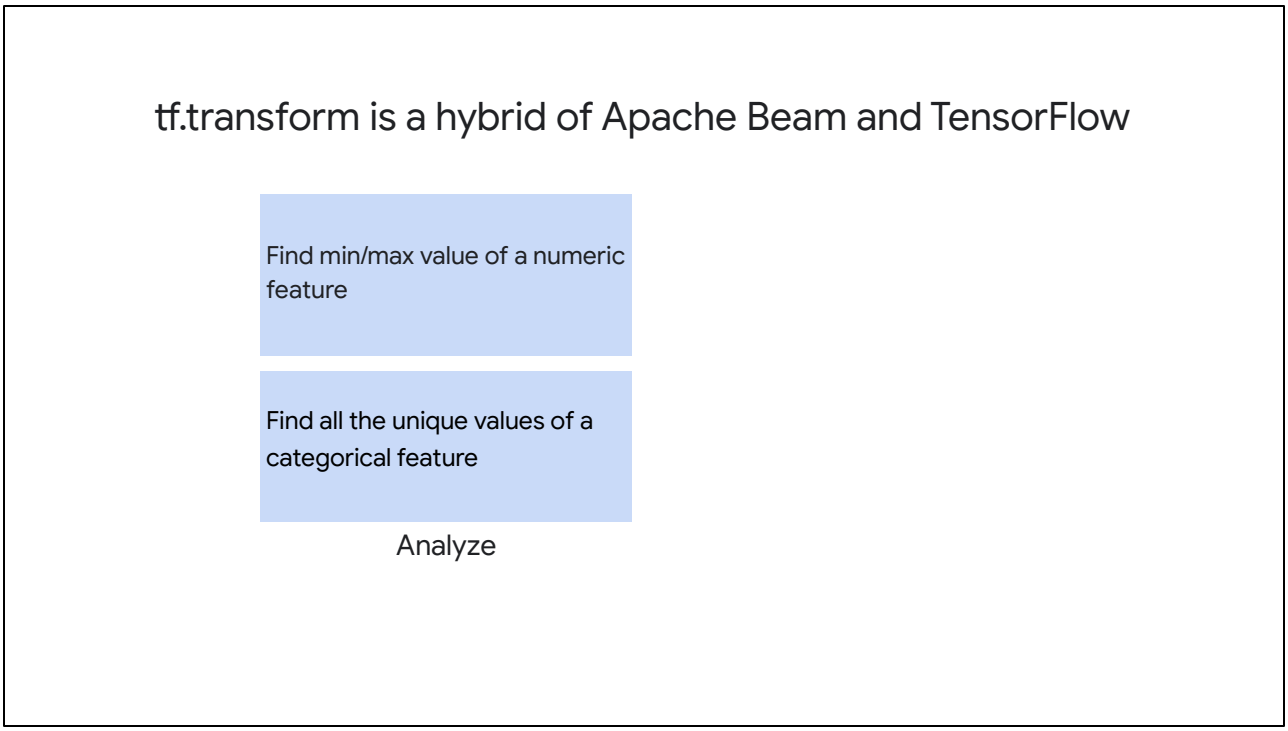
tf.Transform



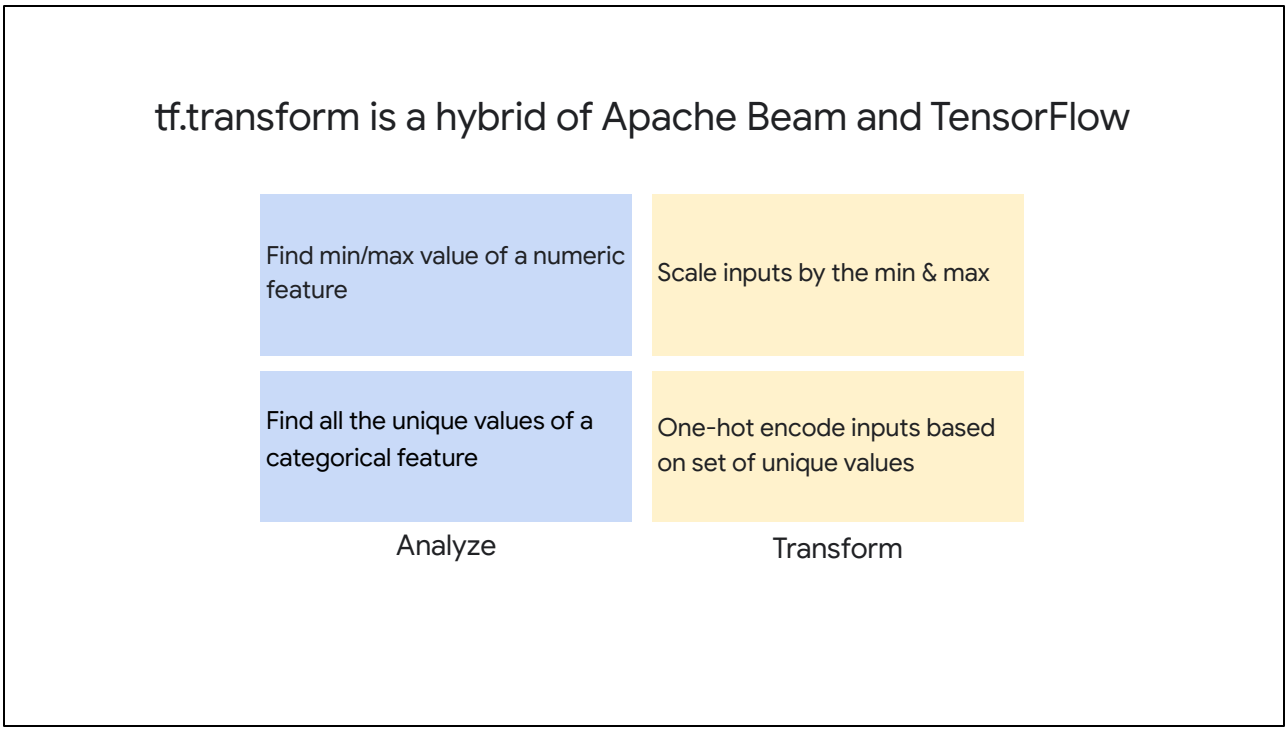
tf.Transform performs a full pass over the data before the user starts their Trainer job.

tf.transform is a hybrid of Apache Beam and TensorFlow

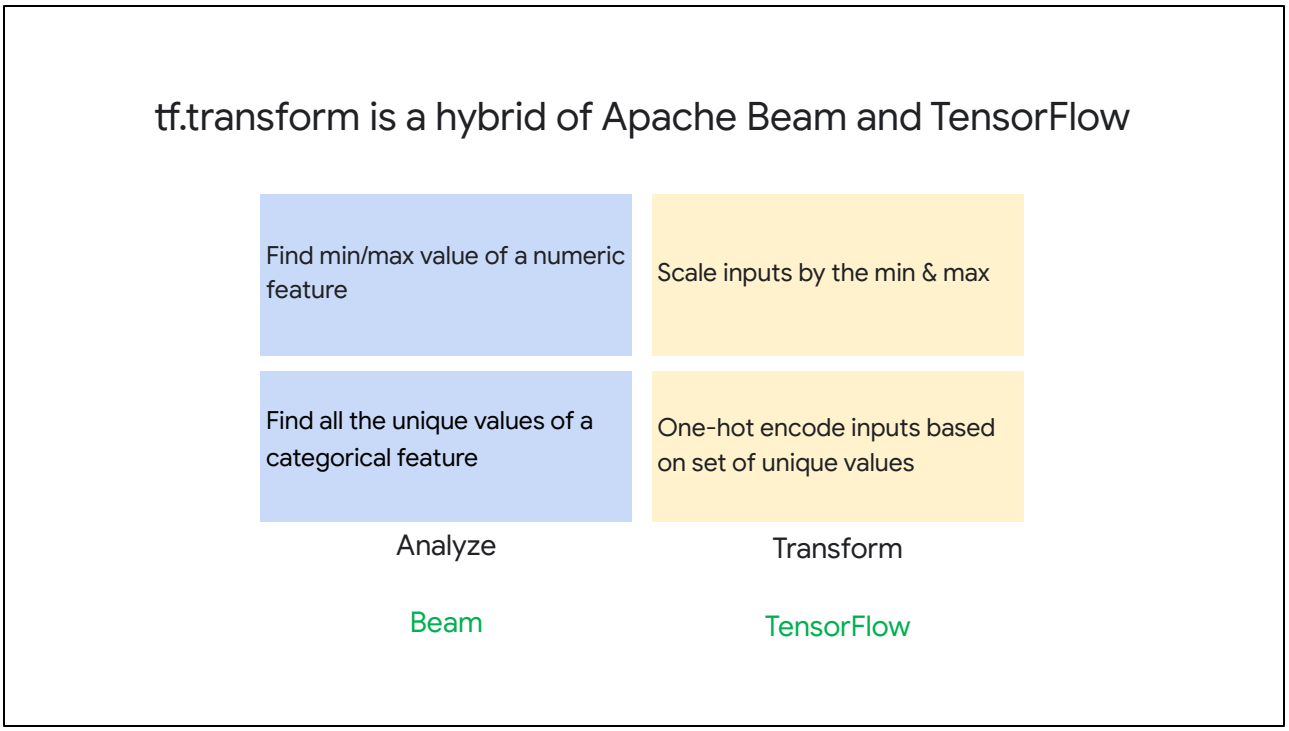
Right -- analysis in Beam, transform in TensorFlow



Right -- analysis in Beam, transform in TensorFlow



Right -- analysis in Beam, transform in TensorFlow



Right -- analysis in Beam, transform in TensorFlow

tf.transform has two phases

Analysis phase (compute min/max/vocab etc. using Beam)

- Executed in Beam while creating training dataset

Transform phase (scale/vocabulary etc. using TensorFlow)

- Executed in TensorFlow during prediction
- Executed in Beam to create training/evaluation datasets

Another way to think about it is that there are two phases.

tf.transform provides two PTransforms

`AnalyzeAndTransformDataset`

- Executed in Beam to create the training dataset.
- Similar in purpose to Scikit-learn's `fit_transform` method

`TransformDataset`

- Executed in Beam to create the evaluation dataset
- The underlying transformations are executed in TensorFlow at prediction time
- Similar in purpose to Scikit-learn's `tranform` method.

There are two PTransforms in tf.transform

First, set up the schema of the training dataset

Let’s look at the analyze phase. Remember that you analyze the training dataset.
You first have to tell Beam what kind of data to expect.

First, set up the schema of the training dataset

```
raw_data_schema = {
    colname : dataset_schema.ColumnSchema(tf.string, ...)
        for colname in 'dayofweek,key'.split(',')
}
```

You do that by setting up a schema.
In the first line, I set up a dictionary called raw_data_schema and add entries for all the string columns.

First, set up the schema of the training dataset

```
raw_data_schema = {
    colname : dataset_schema.ColumnSchema(tf.string, ...)
        for colname in 'dayofweek,key'.split(',')
}
```

TensorFlow type for input column

The string here is the TensorFlow data type.

First, set up the schema of the training dataset

```
raw_data_schema = {
    colname : dataset_schema.ColumnSchema(tf.string, ...)
        for colname in 'dayofweek,key'.split(',')
}
raw_data_schema.update({
      colname : dataset_schema.ColumnSchema(tf.float32, ...)
        for colname in 'fare_amount,pickuplon, … ,dropofflat'.split(',')
})
```

TensorFlow type for input column

float32

I then update the raw_data_schema by adding all the tf.float32 typed columns.

First, set up the schema of the training dataset

```
raw_data_schema = {
    colname : dataset_schema.ColumnSchema(tf.string, ...)
        for colname in 'dayofweek,key'.split(',')
}
raw_data_schema.update({
      colname : dataset_schema.ColumnSchema(tf.float32, ...)
        for colname in 'fare_amount,pickuplon, … ,dropofflat'.split(',')
})
raw_data_metadata =
    dataset_metadata.DatasetMetadata(dataset_schema.Schema(raw_data_schema))
```

TensorFlow type for input column

float32

After this, I have a raw_data_schema has all the columns in the dataset that will be processed by Beam on Dataflow.

First, set up the schema of the training dataset

```
raw_data_schema = {
    colname : dataset_schema.ColumnSchema(tf.string, ...)
        for colname in 'dayofweek,key'.split(',')
}
raw_data_schema.update({
      colname : dataset_schema.ColumnSchema(tf.float32, ...)
        for colname in 'fare_amount,pickuplon, … ,dropofflat'.split(',')
})
raw_data_metadata =
    dataset_metadata.DatasetMetadata(dataset_schema.Schema(raw_data_schema))
```

TensorFlow type for input column

float32

Use the schema to create metadata "template"

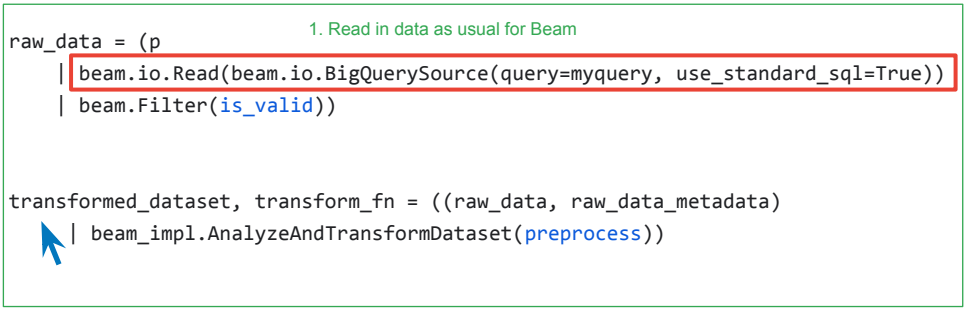
The raw data schema is used to create a metadata template.

Next, run the analyze-and-transform PTransform on training dataset to get back preprocessed training data and the transform function

```
raw_data = (p
    | beam.io.Read(beam.io.BigQuerySource(query=myquery, use_standard_sql=True))
    | beam.Filter(is_valid))


transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
    | beam_impl.AnalyzeAndTransformDataset(preprocess))
```

Next, run the analyze-and-transform PTransform on training dataset to get back preprocessed training data and the transform function
First, do beam.io.Read to read in the training data. This is similar to all the Beam pipelines you saw in the previous module on Beam.

Next, run the analyze-and-transform PTransform on training dataset to get back preprocessed training data and the transform function



Here, I'm reading from BigQuery
Next, filter out the data that you don't want to train with

Next, run the analyze-and-transform PTransform on training dataset to get back preprocessed training data and the transform function

```
raw_data = (p
    | beam.io.Read(beam.io.BigQuerySource(query=myquery, use_standard_sql=True))
    | beam.Filter(is_valid))

transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
    | beam_impl.AnalyzeAndTransformDataset(preprocess))
```

1. Read in data as usual for Beam

2. Filter out data that you don't want to train with

I'm doing that with a function is_valid that I am not showing you on this slide. I'll show you this method later.
Third, take the raw data that you get from reading and filtering and the raw data metatadata that you get from the previous slide and

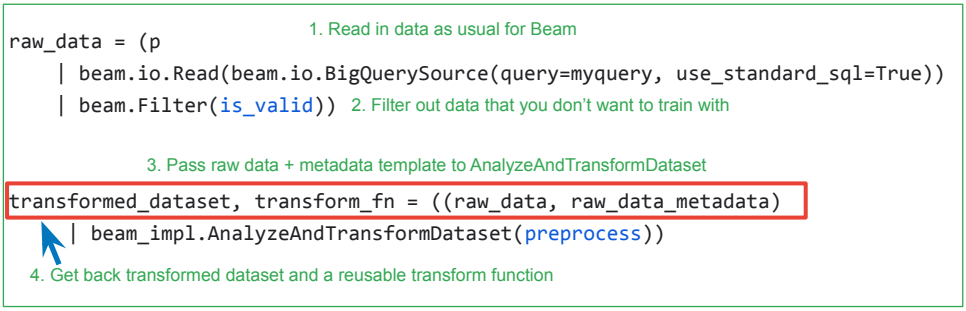
Next, run the analyze-and-transform PTransform on training dataset to get back preprocessed training data and the transform function

```
raw_data = (p
    | beam.io.Read(beam.io.BigQuerySource(query=myquery, use_standard_sql=True))
    | beam.Filter(is_valid))

transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
    | beam_impl.AnalyzeAndTransformDataset(preprocess))
```

1. Read in data as usual for Beam
2. Filter out data that you don't want to train with
3. Pass raw data + metadata template to AnalyzeAndTransformDataset

Pass it to the Analyze And Transform Dataset PTransform.
Beam will execute this transform in a distributed way, and do all the analysis that you told it to do in the method preprocess. I'll show you what this method looks like later.
For now, the is_valid() method and the preprocess() method are executed by Beam on the training dataset to filter it and preprocess it.

Next, run the analyze-and-transform PTransform on training dataset to get back preprocessed training data and the transform function



The preprocessed data comes back in a PCollection that I'm calling transformed_dataset

But notice that the transformations that you carried out in preprocess are saved in the second return value -- transform_fn

This is important.

Write out the preprocessed training data into TFRecords, the most efficient format for TensorFlow

```
transformed_data | tfrecordio.WriteToTFRecord(

                     os.path.join(OUTPUT_DIR,'train'),

                     coder=ExampleProtoCoder(
                          transformed_metadata.schema)
                          )
```

Take the transformed data and write it out.
Here, i am writing it out as TFRecords, which is the most efficient format for TensorFlow.
I can do that by using the WriteToTFRecord PTransform that comes with tf transform.
The files will be sharded automatically.

Write out the preprocessed training data into TFRecords, the most efficient format for TensorFlow

```
transformed_data | tfrecordio.WriteToTFRecord(

                       os.path.join(OUTPUT_DIR,'train'),
                              The filenames will be like train-0003-of-0015
                       coder=ExampleProtoCoder(
                            transformed_metadata.schema)
                            )
```

But notice what schema is being used

Write out the preprocessed training data into TFRecords, the most efficient format for TensorFlow

```
transformed_data | tfrecordio.WriteToTFRecord(

                       os.path.join(OUTPUT_DIR,'train'),
                              The filenames will be like train-0003-of-0015
                       coder=ExampleProtoCoder(
                            transformed_metadata.schema)
                            )   Note that we use the transformed metadata
                                schema here!
```

Not the raw data schema! The transformed schema!
Why?
[pause]
Because, of course, what we are writing out is the transformed data, not the raw data.

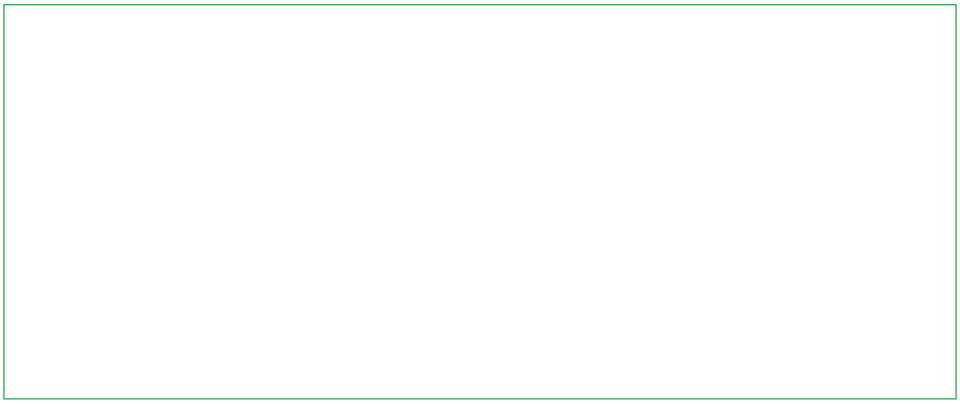
The preprocessing function is restricted to TensorFlow functions

```
transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
     | beam_impl.AnalyzeAndTransformDataset(
                  preprocess))
```

The things you do in preprocess() will get added to the TensorFlow graph, and be executed in TensorFlow during serving

The preprocessing function is the function where transform the input data.
In Beam, it is called as part of the AnalyzeAndTransformDataset
In TensorFlow, the things you do in preprocess() will get called essentially as part of the serving_input_fn in TensorFlow.

The preprocessing function is restricted to functions you can call from TensorFlow graph



You can not call regular python functions, since the preprocess is part of the tensorflow graph during serving.
Let’s look at an example.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):
```

In this example, I am taking a set of inputs and preprocessing them.
What is the data type of ‘inputs’?
[pause]
It is a dictionary whose values are tensors.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):

     result = {}    Create features from the input tensors and put into “results” dict
```

Remember: this is what is returned from the serving_input function and represents the raw data as it is read.
Input functions return features comma labels, and this is the features. And features is a dict.
Tf.transform will take care of converting the data that comes in via PTransform into tensors during the analysis phase.
We take the tensors and use them to create new features, and put these features into a new dictionary.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):

    result = {}     Create features from the input tensors and put into “results” dict

  result['fare_amount'] = inputs['fare_amount']  Pass through
```

The first result, ‘fare_amount’ in my example is passed through unchanged. We take the input tensor and it to the result. No changes.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):

    result = {}     Create features from the input tensors and put into "results" dict

  result['fare_amount'] = inputs['fare_amount']  Pass through

  result['dayofweek'] = tft.string_to_int(inputs['dayofweek'])  vocabulary
```

The next result we want is 'dayofweek'. We want this to be an integer. However, in the input, it is a string like "Thu". So, what we are doing it is that we are asking tf.transform to convert the string that is read (such as "Thu") into an integer (such as 3 or 5). What tf transform will do is to compute the vocabulary of all the possible days of the week in the training dataset during the Analyze phase and use that information to do the string-to-int mapping.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):

     result = {}    Create features from the input tensors and put into “results” dict

   result['fare_amount'] = inputs['fare_amount']  Pass through

   result['dayofweek'] = tft.string_to_int(inputs['dayofweek'])  vocabulary
   ...
   result['dropofflat'] = (tft.scale_to_0_1(inputs['dropofflat']))     scaling
```

Next, we want to scale off the dropofflat into a number that lies between 0 and 1. IN the analysis phase, Tf.transform will compute the min and max of the column and use those values to scale the inputs.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):
    result = {}
  result['fare_amount'] = inputs['fare_amount']
  result['dayofweek'] = tft.string_to_int(inputs['dayofweek'])
  ...
  result['dropofflat'] = (tft.scale_to_0_1(inputs['dropofflat']))
  result['passengers'] = tf.cast(inputs['passengers'], tf.float32)
```

Create features from the input tensors and put into “results” dict

Pass through

vocabulary

scaling

Other TF fns

We can also invoke other TensorFlow functions. In tihs case, I am take the input # of passengers, which happens to be an integer in JSON and casting it to a real-valued number.

The preprocessing function is restricted to functions you can call from TensorFlow graph

```
def preprocess(inputs):

    result = {}    Create features from the input tensors and put into "results" dict

  result['fare_amount'] = inputs['fare_amount']  Pass through

  result['dayofweek'] = tft.string_to_int(inputs['dayofweek'])  vocabulary
  ...
  result['dropofflat'] = (tft.scale_to_0_1(inputs['dropofflat']))    scaling

  result['passengers'] = tf.cast(inputs['passengers'], tf.float32)  Other TF fns

  return result
```

Once all the features have been created and added, we can return the result.

Analyze and Transform happens on the training dataset

```
transformed_dataset, transform_fn = ((raw_data, raw_data_metadata)
     | beam_impl.AnalyzeAndTransformDataset(preprocess))
```

The analyze and transform PTransform happens on the training dataset.
What should happen on the evaluation dataset?

Writing out the eval dataset is similar, except that we reuse the transform function computed from the training data

```
raw_test_data = (p
        | beam.io.Read(beam.io.BigQuerySource(...)
        | 'eval_filter' >> beam.Filter(is_valid))
transformed_test_dataset = (((raw_test_data, raw_data_metadata), transform_fn)
        | beam_impl.TransformDataset())
```

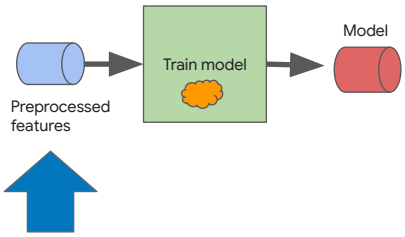
For the evaluation dataset, we carry out pretty much the same Beam pipeline that we did on the training dataset.
There is one big exception, though.
We don't analyze the evaluation dataset. If we are scaling the values, the values in the evaluation dataset will be scaled based on the min & max found in the training dataset.
So, on the evaluation dataset, we just call TransformDataset. This will take care of calling all the things that we did in preprocess! Pretty cool, eh?
Note, however, that the TransformDataset needs as input the transform_fn that was computed on the training data.
That's what makes the magic possible.

Writing out the eval dataset is similar, except that we reuse the transform function computed from the training data

```
raw_test_data = (p
        | beam.io.Read(beam.io.BigQuerySource(...)
        | 'eval_filter' >> beam.Filter(is_valid))
transformed_test_dataset = (((raw_test_data, raw_data_metadata), transform_fn)
        | beam_impl.TransformDataset())

transformed_test_data, _ = transformed_test_dataset
_ = transformed_test_data | tfrecordio.WriteToTFRecord(
          os.path.join(OUTPUT_DIR, 'eval'),
          coder=example_proto_coder.ExampleProtoCoder(
              transformed_metadata.schema))
```

Once we have the transformed dataset
We can write it out just like we wrote out the training dataset.

For training and evaluation, we created preprocessed features using Beam



Created by AnalyzeAndTransformDataset
Or by TransformDataset

We used the transform function to transform the evaluation dataset and we wrote the transformed evaluation data.
For what type of data did we use AnalyzeAndTransformDataset?
[pause]
Right … the training data
And we used TransformDataset for the evaluation data
Even though we created the preprocessed features using Beam, the preprocess method itself consisted solely of TensorFlow functions.

For serving, we need to write out the transformation metadata

```
_ = transform_fn | transform_fn_io.WriteTransformFn(
     os.path.join(OUTPUT_DIR, 'metadata')))
```

The reason these functions needed to be in TensorFlow was that they are part of the prediction graph, so that the end-user can give the model raw data and the model can do the necessary preprocessing.
But how will the model know what functions to call? In order for the model to know what functions to call, we need to save the transform function.
That's what I'm doing here … I'm saving the transform function itself into a directory called metadata, alongside my trained model.
Then we tell the input function to pick up the metadata.

Change input function to read preprocessed features

```
def input_fn(tf_transform_output, transformed_examples_pattern, batch_size):

  return tf.data.experimental.make_batched_features_dataset(
      file_pattern=transformed_examples_pattern,
      batch_size=batch_size,
      features=tf_transform_output.transformed_feature_spec(),
      reader=tf.data.TFRecordDataset,
      label_key=LABEL_KEY,
      shuffle=True).prefetch(tf.data.experimental.AUTOTUNE)
```

Which input function?
Well, all three. First, let's look at the training and evaluation input functions.
They read the preprocessed features

Change input function to read raw features

```
dataset = tf.data.experimental.make_csv_dataset(...)

tft_layer = tf_transform_output.transform_features_layer()

def transform_dataset(data):
  ...
  transformed_features = tft_layer(raw_features)
  ...
  return (transformed_features, data_labels)

return dataset.map(
    transform_dataset,
    num_parallel_calls=tf.data.experimental.AUTOTUNE).prefetch(
        tf.data.experimental.AUTOTUNE)
```

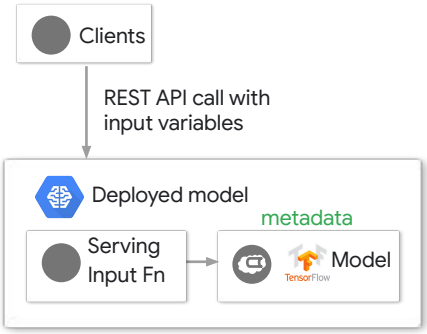
So, notice that i specify that the schema corresponds to the transformed_metadata
Change the training and evaluation input functions to read the preprocessed features.

The serving input function accepts the raw data

```
model.tft_layer = tf_transform_output.transform_features_layer()

@tf.function
def serve_tf_examples_fn(serialized_tf_examples):
  feature_spec = RAW_DATA_FEATURE_SPEC.copy()
  feature_spec.pop(LABEL_KEY)
  parsed_features = tf.io.parse_example(serialized_tf_examples, feature_spec)
  transformed_features = model.tft_layer(parsed_features)
  outputs = model(transformed_features)
  classes_names = tf.constant([['0', '1']])
  classes = tf.tile(classes_names, [tf.shape(outputs)[0], 1])
  return {'classes': classes, 'scores': outputs}

concrete_serving_fn = serve_tf_examples_fn.get_concrete_function(
    tf.TensorSpec(shape=[None], dtype=tf.string, name='inputs'))
signatures = {'serving_default': concrete_serving_fn}

model.save(output_dir, save_format='tf', signatures=signatures)
```

The serving input function accepts the raw data.
So, here, I'm passing in the raw data metadata
Well, the raw data alone isn't enough. We could also have arbitrary TensorFlow functions in the preprocessing code.
Those operations are stored in saved_model.pb
But again, notice the nice tensorflow transform helper function.
Build_parsing_transforming_serving_input_fn
Parse the json according to the raw schema
Transform the raw data based on the TensorFlow operations in saved_model.pb
Then send it along to the model!

The model graph includes the preprocessing code



This happens because the model reads the metadata and includes the preprocessing code ….
So that's how tensorflow transform works. Let's use it on our taxifare prediction problem.

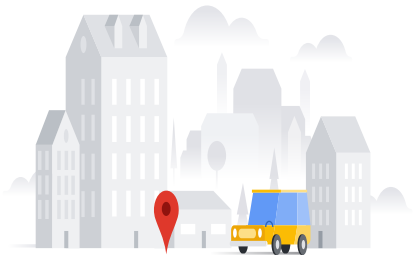
Lab

Exploring TF Transform

In this lab, we will look at how to use TensorFlow transform.
We will write a Beam pipeline to analyze and transform the training data.
In that same Beam pipeline, we'll also transform the evaluation data and save the transform function.
We'll modify the training and evaluation input functions to read these preproprocessed files.
Then, we'll train the model as normal.
Having done that, we'll be able to do preprocessing at scale over large datasets during training using Dataflow
And do preprocessing efficiently, as part of the model graph, in TensorFlow during serving.

Lab

1. Preproccess data and engineer new features using TfTransform
2. Create and deploy Apache Beam pipeline
3. Use processed data to train taxifare model locally then serve a prediction



The objectives of the first lab are to:

1. Define features, the Label, and default values
2. Create an input pipeline using tf.data
3. Create ML models using Keras
4. Visualize the DNN model layers
5. Train the model/visualize the loss curve
6. Incorporate temporal features
7. Incorporate Geolocation Features with Bucketization and Feature Crosses
8. Feature Cross Temporal Features
9. Create and test a prediction model

Feature Crosses - TensorFlow Playground - Optional

Welcome to Feature Crosses.

Agenda

Building a Feature Cross

Embeddings from Feature Crosses

Next you're going to look at representing features with some examples.

Can you draw a line that separates these two classes?

You may recall using these diagrams to explain what neural networks were? You could think of the blue dots as, maybe customers who buy a particular phone.

Can you draw a line that separates these two classes?

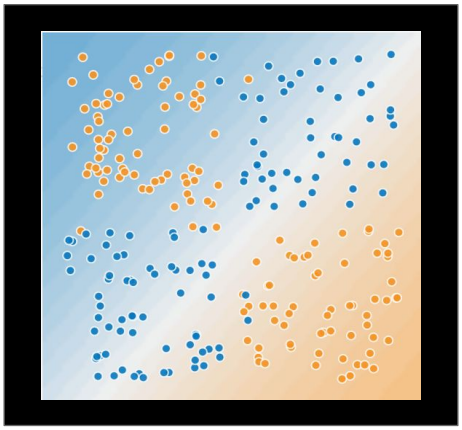
You could think of the blue dots as, maybe customers who buy a particular phone. The yellow dots could be seen as customers who don't buy the phone. Perhaps the x axis is the time since this customer last bought a phone and maybe the y axis is the income level of the customer.
Essentially, people buy the product if it has been a long time since they bought a phone, and they are relatively wealthy.

This is a linear problem

It might have a little bit of error (not perfectly separable) but a linear model is probably pretty good here.

The blue dots and yellow dots are linearly separable by the green line.

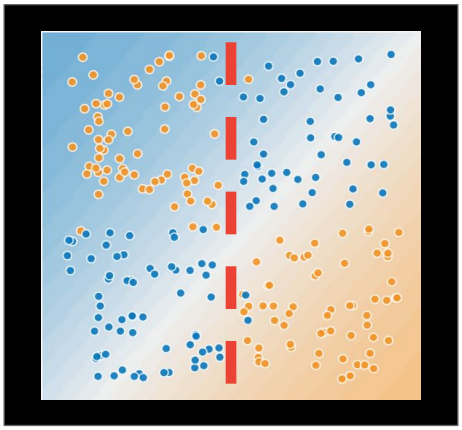
How about this? Is it a linear problem?



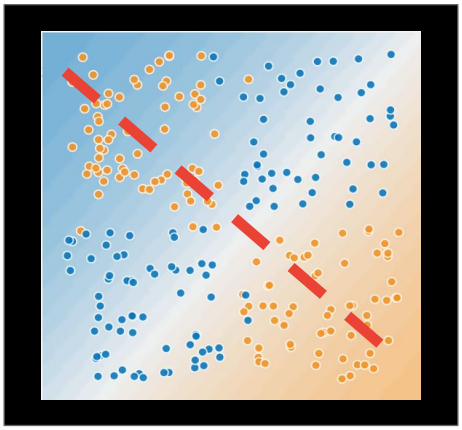
But, can you still use a linear model if your data looks like this?

It seems that you can not draw a line that manages to separate the blue dots from the yellow dots.

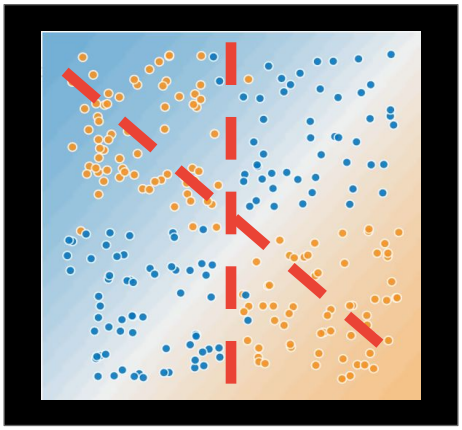
How about this? Is it a linear problem?



How about this? Is it a linear problem?



How about this? Is it a linear problem?

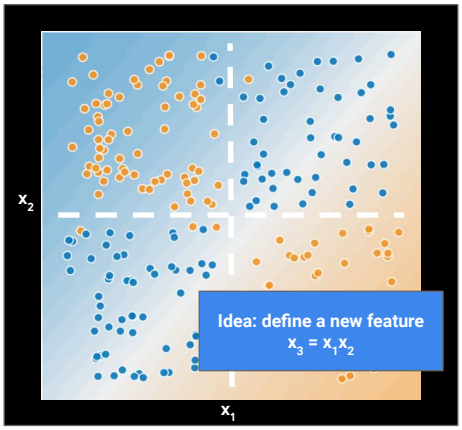


Wherever you draw your line, there are blue points on either side of the line. The data are not linearly separable.

This would mean you cannot use a linear model.

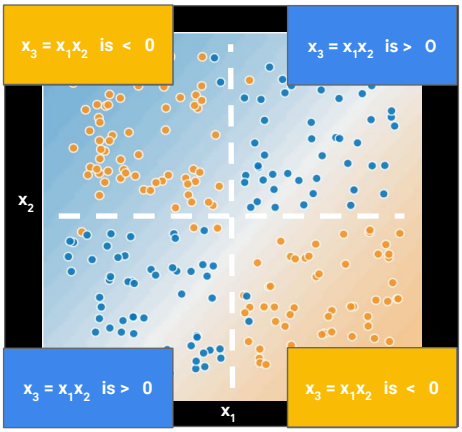
Can we be a bit more specific about what is meant by “linear model”?

How about this? Is it a linear problem?



Define a new feature, x3, as the product of x1 and x2.
So, how does this help?

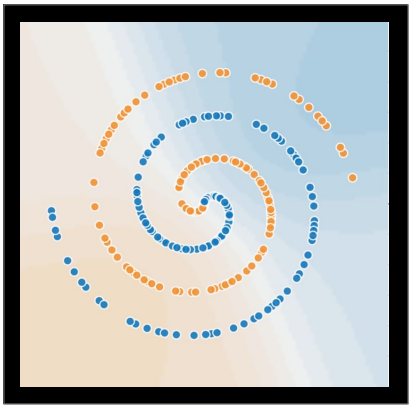
How about this? Is it a linear problem?



When x1 or x2 is negative and the other one is positive.

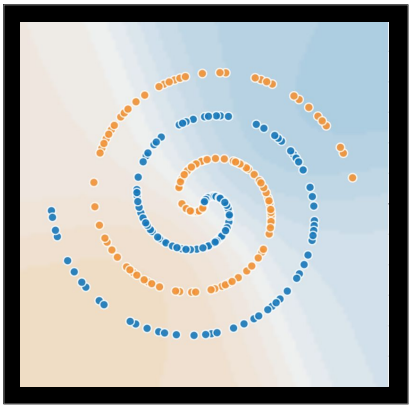
Now we have x3. Can you see how the addition of x3 makes this solvable via a linear model?

Can a linear model work for this problem?



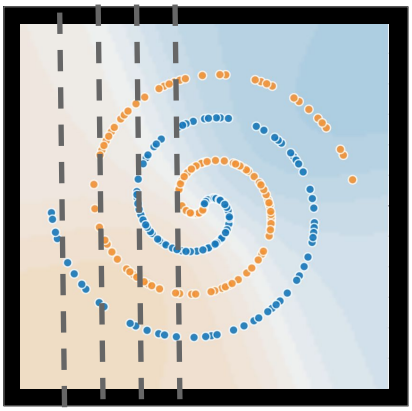
In this example you are presented with a more complex problem. Consider whether a linear model would help.

What if we discretize x1 and x2 and then multiply?



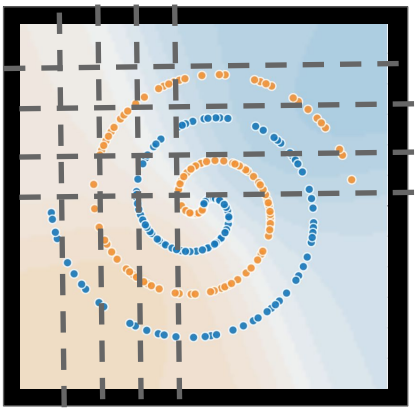
What if you were to discretize the x1 axis by drawing not just one white line, but lots of these black lines?

What if we discretize x1 and x2 and then multiply?



Do the same for the x2 axis, by drawing a whole bunch of black lines.

What if we discretize x1 and x2 and then multiply?



You would now have discretized the x1 axis and the x2 axis.
When you drew two white lines, you ended up with four quadrants. What about now?

If you have M vertical lines and N horizontal lines, we will end up with M times N grid cells, right?

Consider what this looks like when you discretize x1 and x2 and then multiply

Playground Lab

Use feature crosses to create a good classifier

In this lab, you will use use the TensorFlow playground to develop an intuitive understanding of feature crosses.

https://goo.gl/2NUCAF

https://goo.gl/ivd4x4

1. What's the best performance you can get?
2. Which feature crosses help the most?
3. Does the model output surface look like a linear model?

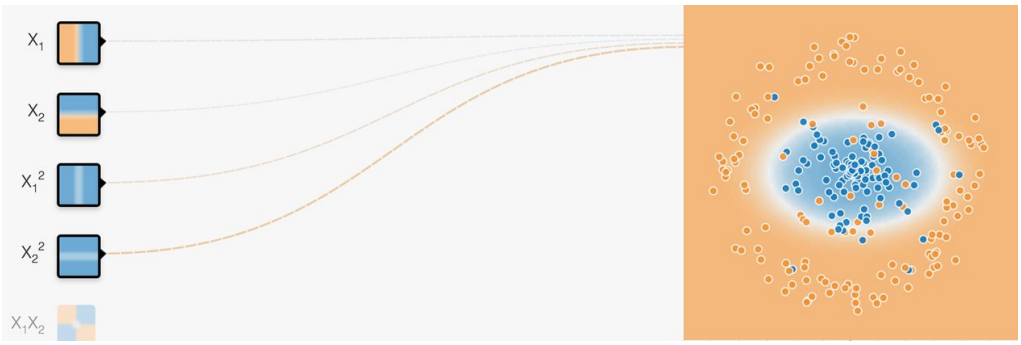
In this lab, you will use use the TensorFlow playground to develop an intuitive understanding of feature crosses.
Click on the links and try to add new features while continuing to use a linear model.

What's the best performance you can get?
Which feature crosses help the most?
Does the model output surface look like a linear model?

http://playground.tensorflow.org/#activation=relu&batchSize=5&dataset=xor&regDataset=reg-plane&learningRate=0.01&regularizationRate=0&noise=35&networkShape=&seed=0.22297&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

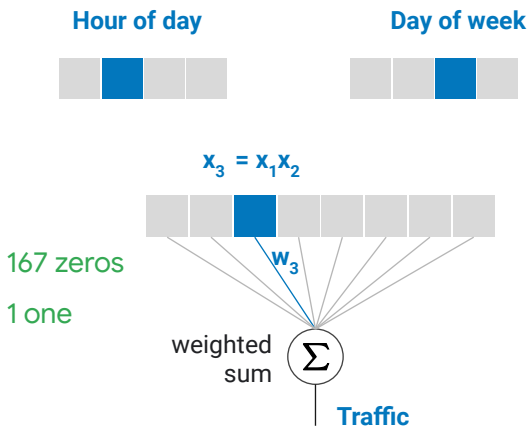
http://playground.tensorflow.org/#activation=relu&batchSize=5&dataset=circle&regDataset=reg-plane&learningRate=0.01&regularizationRate=0&noise=35&networkShape=&seed=0.92217&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

The linear decision boundary gets transformed into the curve in the original coordinate space



The linear decision boundary gets transformed into the curve in original coordinate space. By using feature crosses, you get the benefits of non-linearity even while using linear models, you get to enjoy the benefits of convex optimization *and* non-linearity.

Feature Crosses lead to sparsity



For any particular row of your input dataset, how many nodes in x3 are “lit up”?

Just one.

Do you see why?

Every label, every observation of table, is taken at a specific time. That corresponds to a specific hour on a specific day of the week. So, 3pm in the hour-of-day input and Wednesday in the day-of-week input.

Feature Cross these. And what do you have? You have one input node, the input node that corresponds to 3pm on Wednesday will be 1. All the other input nodes for x3 will be zero.

The input, therefore, will consist of 167 zeros and one 1.

When you do a feature cross, the input is very, very sparse. TensorFlow will give us easy tools to deal with this.

Quiz: Which of these is a good feature cross?

Different cities in California have markedly different housing prices. Suppose you must create a model to predict housing prices. Which of the following sets of features or feature crosses could learn city-specific relationships between house characteristic and housing price?

a) Three separate binned features: [binned latitude], [binned longitude], [binned roomsPerPerson]

a) Two feature crosses: [binned latitude X binned roomsPerPerson] and [binned longitude X binned roomsPerPerson]

a) One feature cross: [binned latitude X binned longitude X binned roomsPerPerson]

a) One feature cross: [latitude X longitude X roomsPerPerson]

Quiz: Which of these is a good feature cross?

Different cities in California have markedly different housing prices. Suppose you must create a model to predict housing prices. Which of the following sets of features or feature crosses could learn city-specific relationships between house characteristic and housing price?

a) Three separate binned features: [binned latitude], [binned longitude], [binned roomsPerPerson]

a) Two feature crosses: [binned latitude X binned roomsPerPerson] and [binned longitude X binned roomsPerPerson]

a) One feature cross: [binned latitude X binned longitude X binned roomsPerPerson]

a) One feature cross: [latitude X longitude X roomsPerPerson]

A. No. Binning is good because it enables the model to learn nonlinear relationships within a single feature. However, a city exists in more than one dimension, so learning city-specific relationships requires crossing latitude and longitude.

B. No. Binning is a good idea; however, a city is the conjunction of latitude and longitude, so separate feature crosses prevent the model from learning city-specific prices.

C. Yes. Crossing binned latitude with binned longitude enables the model to learn city-specific effects of roomsPerPerson. Binning prevents a change in latitude producing the same result as a change in longitude. Depending on the granularity of the bins, this feature cross could learn city-specific or neighborhood-specific or even block-specific effects.

D. No. In this example, crossing real-valued features is not a good idea. Crossing the real value of, say, latitude with roomsPerPerson enables a 10% change in one feature (say, latitude) to be equivalent to a 10% change in the other feature (say, roomsPerPerson). This is not even possible with TensorFlow -- crossing is only possible with categorical or discretized columns.

Playground Lab

Too much of a good thing

https://goo.gl/ofiHCT

In this lab, you will use use the TensorFlow playground to develop an intuitive understanding of the limits of feature crosses.

1. Is the model behavior surprising? What's the issue?
2. Try removing the cross-product features. What happens?
3. Does performance improve?

In this lab, you will use use the TensorFlow playground to develop an intuitive understanding of the limits of feature crosses.

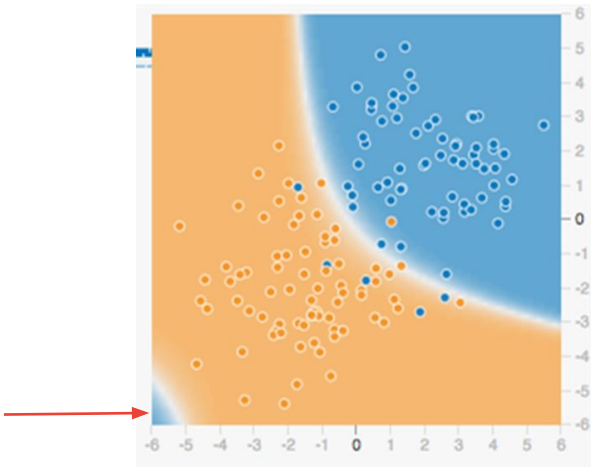
Click on this link and launch the training.

Is the model behavior surprising? Be specific. What is strange? What's the issue?

Now try to removing the cross-product features. What happens?

Does performance improve? Can you explain what is going on?

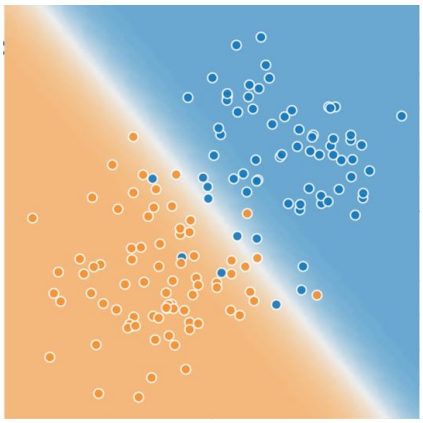
Lab: Too much of a good thing



In this model, the decision boundary looks crazy. In particular, there's a region in the bottom left that's hinting towards blue, even though there's no visible support for that in the data.

Tensorflow playground uses random starting points, so your result might be different.

After removing the feature crosses



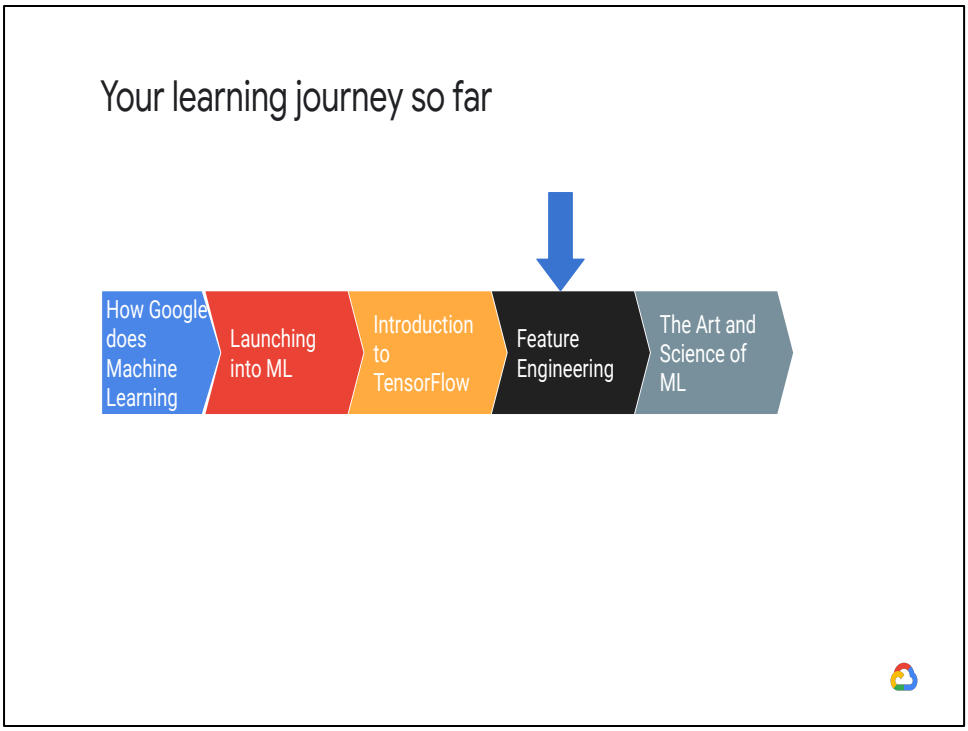
Removing all the feature crosses gives a more sensible model -- there is no longer a curved boundary suggestive of overfitting.

After 1,000 iterations, test loss should be a slightly lower value than when the feature crosses were used (although your results may vary a bit, depending on the data set).

The data in this exercise is basically linear data plus noise. If you use a model that is too complicated, such as one with too many crosses, you give it the opportunity to fit to the noise in the training data

You can often diagnose this by looking at how the model performs on test data.

This explains why L1 regularization can be such a good thing. What L1 regularization does is to zero out the weights of a feature if necessary, in other words, the impact of L1 regularization is to remove features.



And with that, we come to the end of the fourth chapter of this specialization.

cloud.google.com