Google Cloud

Day 2

Introduction

Launching into ML

Welcome to the second chapter in the specialization where we launch into machine learning.
In this chapter you'll get foundational ML knowledge so that you understand the terminology that we use throughout the specialization. You will also learn practical tips and pitfalls from ML practitioners here at Google and walk away with the code and the knowledge to bootstrap your own ML models.

Machine Learning with TensorFlow on GCP

How Google does Machine Learning

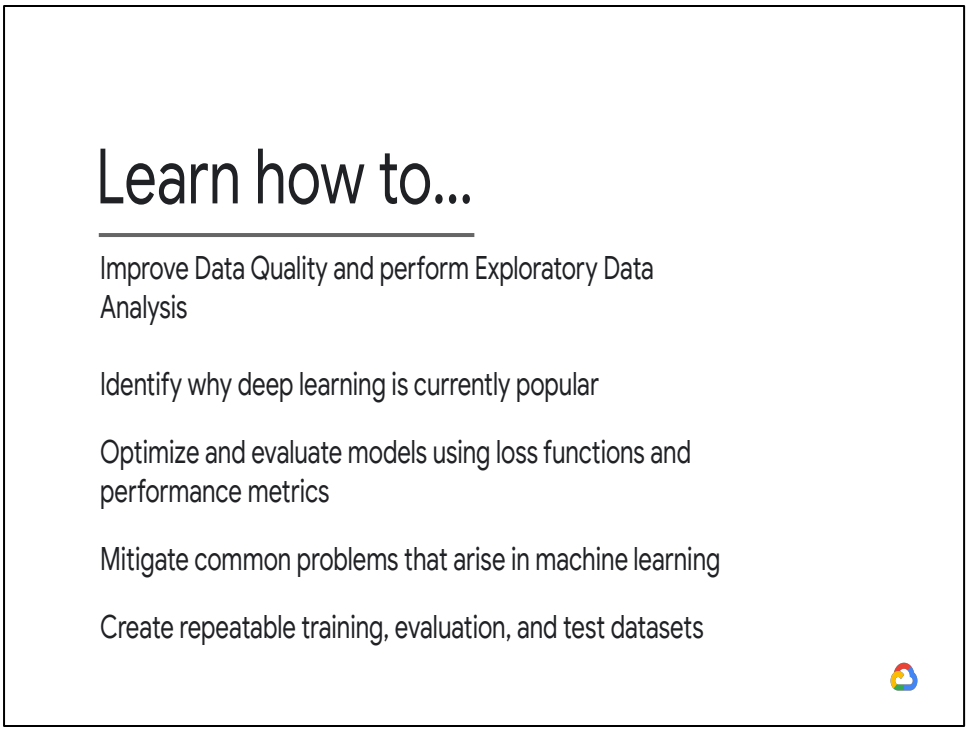
Launching into ML

Introduction to TensorFlow

Feature Engineering

The Art and Science of ML

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



In this course, you will learn about the different types of machine learning models, and how the history of machine learning has led to this point, where deep learning models are so popular.

You'll begin with looking at data - specifically how to improve data quality and how to perform exploratory data analysis.

The training of a deep learning model usually starts with random weights. How do you initialize these weights, and how do you change those weights so that the model learns? You learn how to optimize models using loss functions and evaluate the models using performance metrics.

As you learn how model training and evaluation work, you will also learn about the common problems that can happen when you do machine learning and how to mitigate (how to reduce the incidence of) those problems.

One of the most common problems that can happen is a lack of generalization. When you create an ML model, and it works well in your experiments, but then fails to perform well in production, the failure point will often involve how you created the ML dataset. You will learn why you often need three identically distributed datasets, and how to create them in a repeatable way.

Creating an ML dataset is a practical skill that you do not want to shortchange. Give yourself time to absorb the lessons.

Agenda

Introduction

Improve Data Quality

Exploratory Data Analysis

ML in Practice

Generalization and Sampling

Optimization

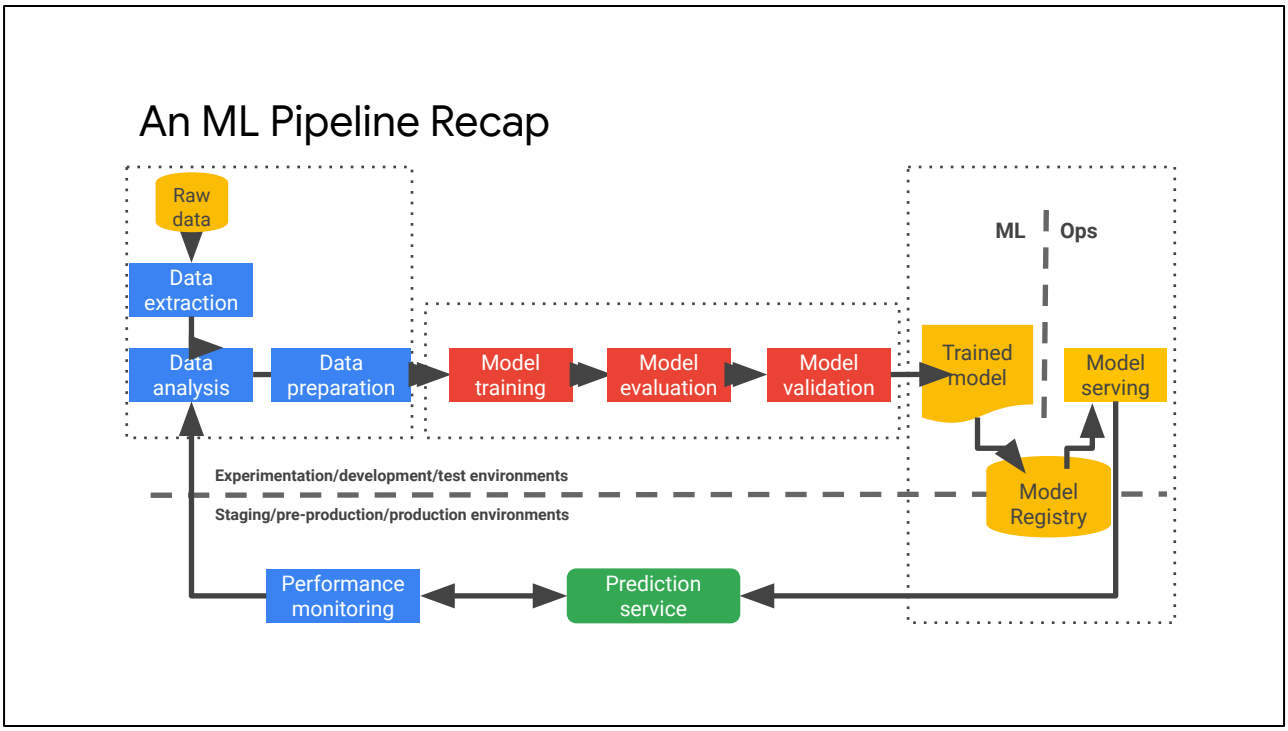
Summary

Google Cloud

Improve Data Quality: An Introduction

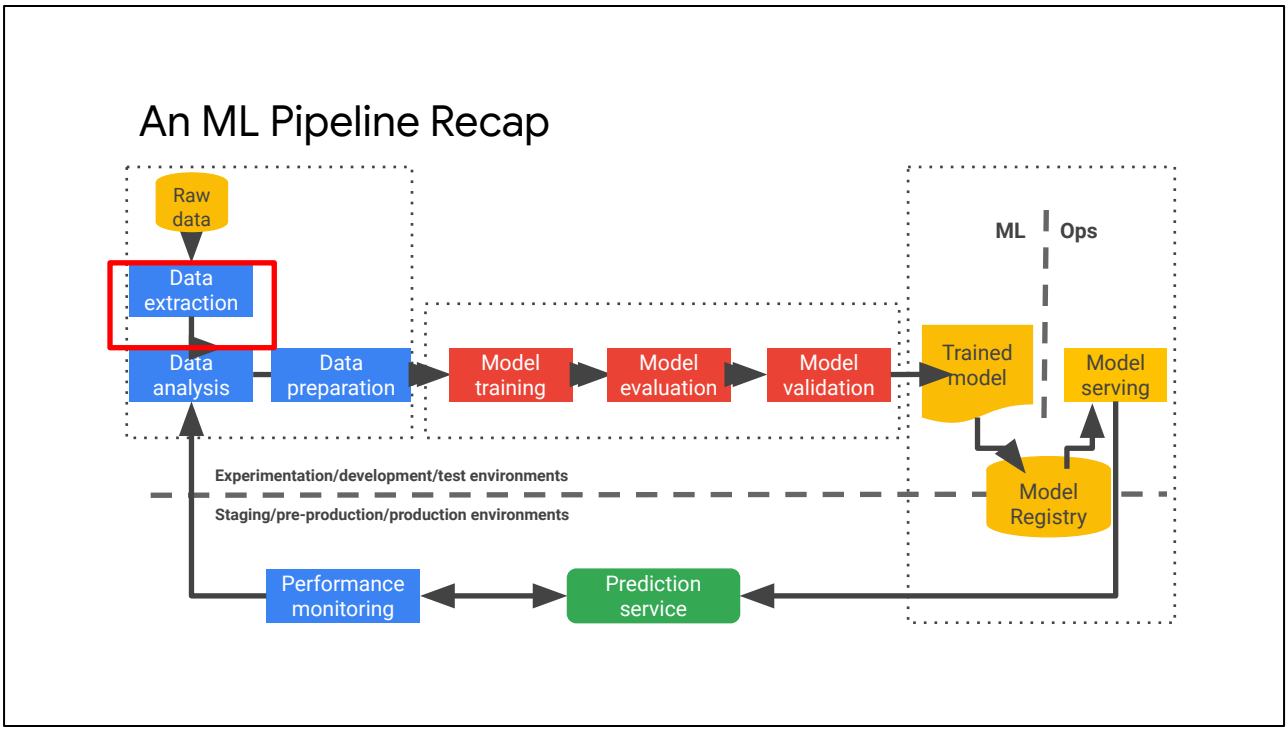
Machine Learning Phases

In the course, you will learn that there are two phases in machine learning, a training phase and an inference phase. You will see that an ML problem can be thought of as being all about data.



In any ML project, after you define the business use case and establish the success criteria, the process of delivering an ML model to production involves the following steps. These steps can be completed manually or can be completed by an automated pipeline:

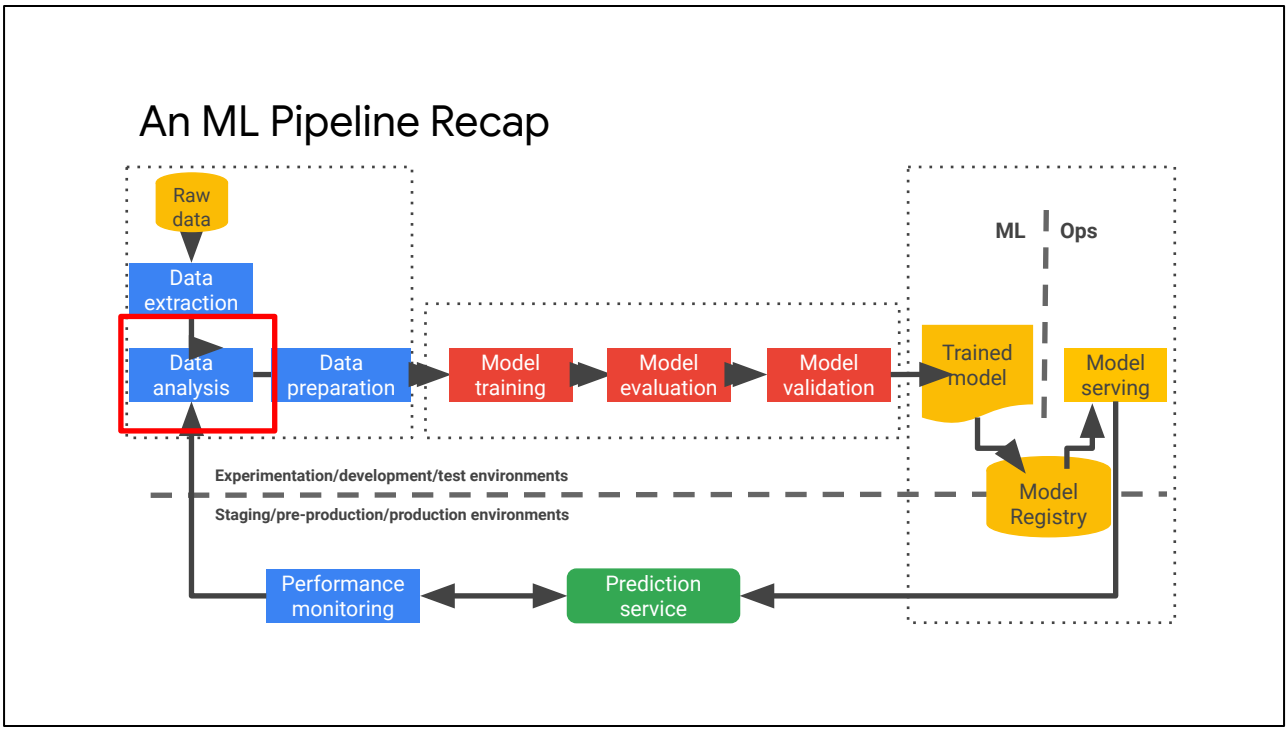
The first three steps deal with data. We can assess the quality of our data in these three steps.



In data extraction, you retrieve data from various sources - those sources can be “streaming in real-time” or “batch”. For example, you may extract data from a customer relationship management system or CRM to analyze customer behavior.

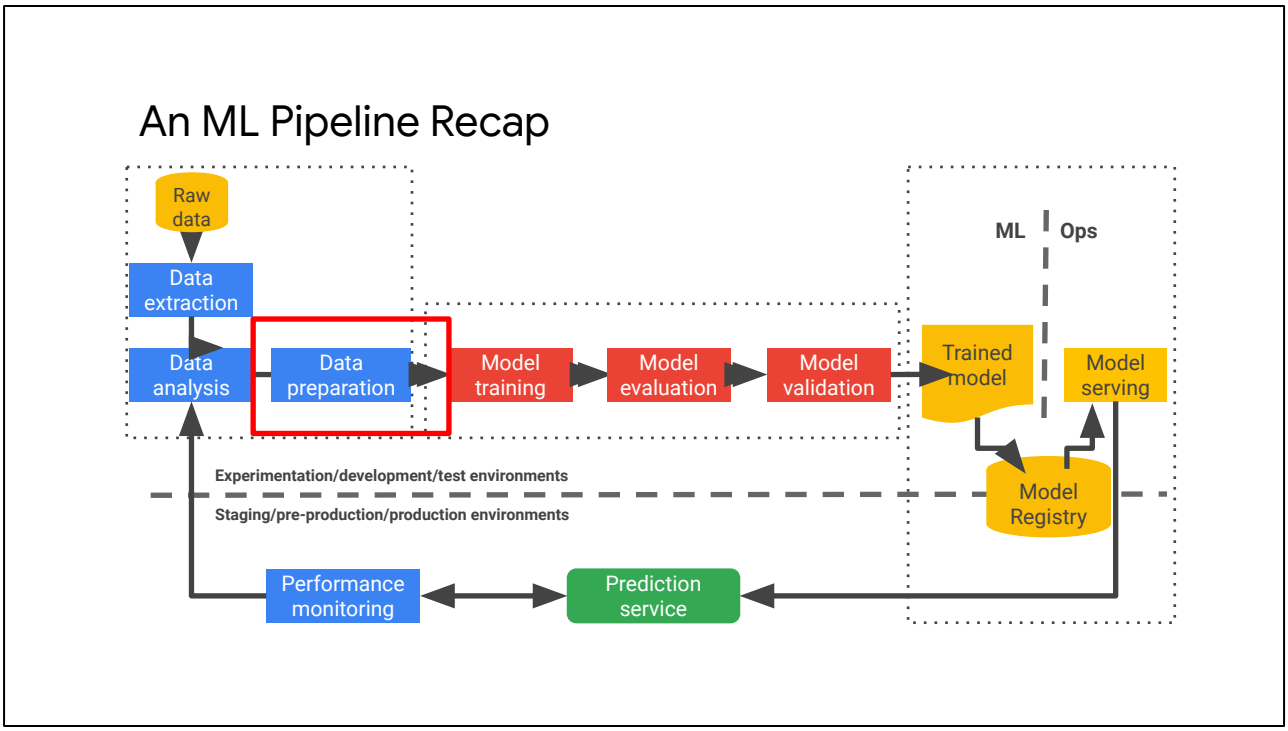
This data may be “structured”, where it is in a given format such as a .csv, .txt. JSON, or . XML format. Or, you may have “unstructured” source data - where you may have images of your customers or text comments from your chat sessions with your customers. Or, you may have to extract “streaming” data from your company’s transportation vehicles that are equipped with sensors that transmit data in real-time.

Other examples of "unstructured data" may include books, journals, documents, metadata, health records, audio, and video.



In data analysis, you analyze the data you've extracted. For example, you can use Exploratory Data Analysis (or EDA), which involves using graphics and basic sample statistics to get a feeling for what information might be obtainable from your dataset.

You look at various aspects of the data, such as outliers or anomalies, trends, and data distributions -- all while attempting to identify those features that can aid in increasing the predictive power of your machine learning model.



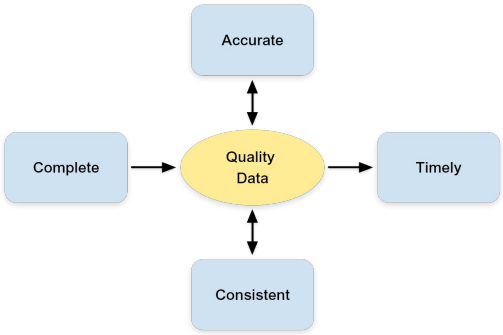
So, after you've extracted and analyzed your data, the next step in the process is data preparation.

Data preparation includes data transformation, which is the process of changing (or converting) the format, structure, or values of data you've extracted into another format or structure.

There are many ways to prepare or "transform" data for a machine learning model. For example, you may need to perform data cleansing – where you need to remove superfluous and repeated data records from raw data. Or, you may need to alter data types – where a data feature was "mistyped" and you need to convert it. Or, you may need to convert categorical data to numerical data. Most ML models require categorical data to be in a numerical format. But some models work either with numeric or categorical features, while others can handle mixed-type features.

Attributes related to the Data Quality

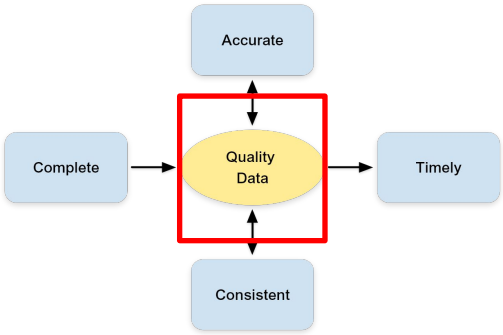
1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data



As a first step toward determining their data quality levels, organizations typically perform data asset inventories in which the relative accuracy, uniqueness and validity of data is measured. The established baseline ratings for data sets can then be compared against the data in systems on an ongoing basis to help identify new data quality issues so they can be resolved.

Attributes related to the Data Quality

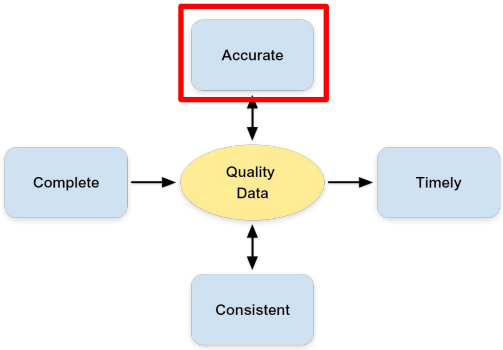
1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data



Let’s look at the attributes related to data quality.

Attributes related to the Data Quality

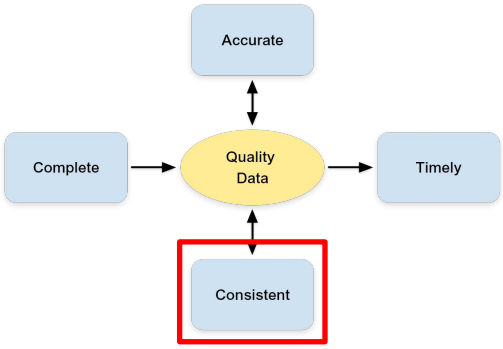
1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data



Data accuracy relates to whether the data values stored for an object are the correct values. For example, does the data match up with the real world object or event it describes, enabling correct conclusions to be drawn from it?

Attributes related to the Data Quality

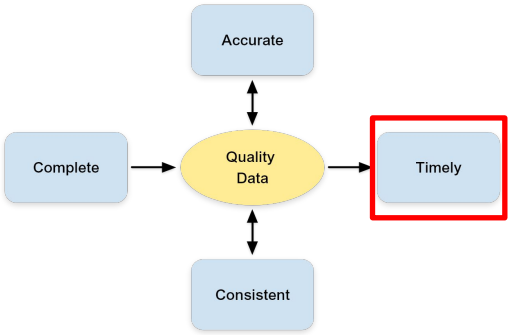
1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data



To be correct, data values must be the right value and must be represented in a consistent and unambiguous form. For example, is the given dataset consistent and correlative with different representations of the same information across multiple datasets?

Attributes related to the Data Quality

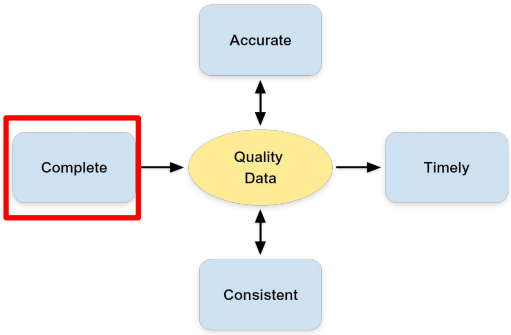
1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data



Timeliness can be measured as the time between when information is expected and when it is readily available for use. For example, how long is the time difference between data capture and the real world event being captured?

Attributes related to the Data Quality

1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data

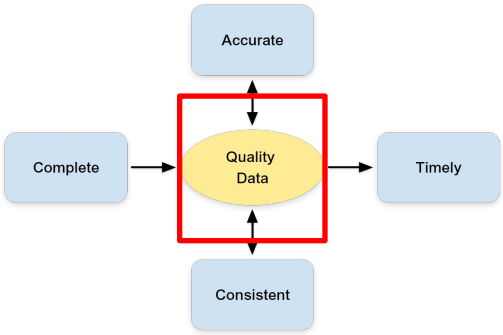


Data completeness relates to whether all the intended data being produced in the data set is complete -- or is any of of the data missing?

Let's look at some examples.

Attributes related to the Data Quality

1. Accuracy of Data
2. Consistency of Data
3. Timeliness of Data
4. Completeness of Data



So, what are ways to improve data quality?

Ways to Improve Data Quality

1. Resolve Missing Values
2. Convert the Date feature column to Datetime Format
3. Parse date/time features
4. Remove unwanted values
5. Convert categorical columns to “one-hot encodings”

You can:
Resolve missing values.
Convert date/time features to a date time format if it is not in the correct format.
You can also parse date/time features to get temporal features that allow you to create more insight into your data.

You can remove unwanted values from a feature column.
AND
You can convert categorical feature columns to “one-hot encodings”.

Missing Values

Let's show the null values for all features in the DataFrame.

```
In [10]: df_transport.isnull().sum()
```

```
Out[10]: Date          2
         Zip Code      2
         Model Year    2
         Fuel          3
         Make          3
         Light_Duty    3
         Vehicles      3
         dtype: int64
```

Missing values can skew your data.

Missing Values

Let's show the null values for all features in the DataFrame.

```
In [10]: df_transport.isnull().sum()

Out[10]: Date          2
         Zip Code      2
         Model Year    2
         Fuel          3
         Make          3
         Light_Duty    3
         Vehicles      3
         dtype: int64
```

In this example, Date, Zip Code, and Model Year have two rows with missing values, while

Missing Values

Let's show the null values for all features in the DataFrame.

```
In [10]: df_transport.isnull().sum()

Out[10]: Date          2
         Zip Code      2
         Model Year    2
         Fuel          3
         Make          3
         Light_Duty    3
         Vehicles      3
         dtype: int64
```

Fuel, Make, Light_Duty, and Vehicles shows three rows with missing values.

Missing Values

```
In [11]: print (df_transport['Date'])
         print (df_transport['Date'].isnull())

         0       10/1/2018
         1       10/1/2018
         2             NaN
         3       10/1/2018
         4       10/1/2018
                    ...
         994      6/7/2019
         995      6/8/2019
         996      6/9/2019
         997     6/10/2019
         998     6/11/2019
         Name: Date, Length: 999, dtype: object
         0       False
         1       False
         2        True
         3       False
```

We can pick one column “Date” and see that it has a NaN (or not a number) on index row 3. We can confirm that by seeing the row 2 is also”true” -- there is a missing value.

Missing Values

```
In [14]: print ("Rows     : " ,df_transport.shape[0])
         print ("Columns  : " ,df_transport.shape[1])
         print ("\nFeatures : \n" ,df_transport.columns.tolist())
         print ("\nUnique values :  \n",df_transport.nunique())
         print ("\nMissing values :  ", df_transport.isnull().sum().values.sum())
```

```
Rows     :  999
Columns  :  7

Features :
 ['Date', 'Zip Code', 'Model Year', 'Fuel', 'Make', 'Light_Duty', 'Vehicles']

Unique values :
 Date          248
Zip Code         6
Model Year      15
Fuel             8
Make            43
Light_Duty       3
Vehicles       210
dtype: int64

Missing values :   18
```

We can run code

Missing Values

```
In [14]: print ("Rows     : " ,df_transport.shape[0])
         print ("Columns  : " ,df_transport.shape[1])
         print ("\nFeatures : \n" ,df_transport.columns.tolist())
         print ("\nUnique values :  \n",df_transport.nunique())
         print ("\nMissing values :  ", df_transport.isnull().sum().values.sum())
```

```
Rows     :  999
Columns  :  7

Features :
 ['Date', 'Zip Code', 'Model Year', 'Fuel', 'Make', 'Light_Duty', 'Vehicles']

Unique values :
 Date          248
Zip Code         6
Model Year      15
Fuel             8
Make            43
Light_Duty       3
Vehicles       210
dtype: int64

Missing values :   18
```

.. to show us all the unique and missing values for our features.

Missing Values = “Messy Data”

```
[5]: df_transport = pd.read_csv('../data/transport/untidy_vehicle_data.csv')
     df_transport.head() # Output the first five rows.
```

[5]:

	Date	Zip Code	Model Year	Fuel	Make	Light_Duty	Vehicles
0	10/1/2018	90000	2006	Gasoline	OTHER/UNK	NaN	1.0
1	10/1/2018	NaN	2014	Gasoline	NaN	Yes	1.0
2	NaN	90000	NaN	Gasoline	OTHER/UNK	Yes	NaN
3	10/1/2018	90000	2017	Gasoline	OTHER/UNK	Yes	1.0
4	10/1/2018	90000	<2006	Diesel and Diesel Hybrid	OTHER/UNK	No	55.0

Indeed, model year and several other features have missing values. This is an example of messy or “untidy data”.

Date and Time

```
In [6]: df_transport.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 7 columns):
Date           997 non-null object
Zip Code       997 non-null object
Model Year     997 non-null object
Fuel           996 non-null object
Make           996 non-null object
Light_Duty     996 non-null object
Vehicles       996 non-null float64
dtypes: float64(1), object(6)
memory usage: 54.8+ KB
```

Another example of ‘untidy” or “messy’ data is when a feature is loaded with the wrong format. In this example, date is listed as a non-null object.

Convert Date and Time

Convert the Date Feature Column to a Datetime Format

The date column is indeed shown as a string object. We can convert it to the datetime datatype with the to_datetime() function in Pandas.

In [19]:
```
# TODO 2a
df_transport['Date'] =  pd.to_datetime(df_transport['Date'],
                                       format='%m/%d/%Y')
```

In [20]:
```
# TODO 2b
df_transport.info() # Date is now converted.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 7 columns):
Date           999 non-null datetime64[ns]
Zip Code       999 non-null object
Model Year     999 non-null object
Fuel           999 non-null object
Make           999 non-null object
Light_Duty     999 non-null object
Vehicles       999 non-null float64
dtypes: datetime64[ns](1), float64(1), object(5)
memory usage: 54.8+ KB
```

Date is not a non-null object and should be converted to a date-time data type. We can convert the date feature to the datetime data type with the to_datetime() function in Pandas.

Parse Date

Let's parse Date into three columns, e.g. year, month, and day.

```
In [21]: df_transport['year'] = df_transport['Date'].dt.year
         df_transport['month'] = df_transport['Date'].dt.month
         df_transport['day'] = df_transport['Date'].dt.day
         #df['hour'] = df['date'].dt.hour - you could use this if your
         #df['minute'] = df['date'].dt.minute - you could use this if y
         df_transport.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 10 columns):
Date          999 non-null datetime64[ns]
Zip Code      999 non-null object
Model Year    999 non-null object
Fuel          999 non-null object
Make          999 non-null object
Light_Duty    999 non-null object
Vehicles      999 non-null float64
year          999 non-null int64
month         999 non-null int64
day           999 non-null int64
dtypes: datetime64[ns](1), float64(1), int64(3), object(5)
memory usage: 78.2+ KB
```

You should also consider parsing the date feature into three distinct feature columns, e.g. year, month, and day. This will allow you to look at the seasonality of your data to spot trends and also perform time-series related predictions.

Unwanted Characters - Model Year

Let's investigate a bit more of our data by using the .groupby() function.

In [9]:
```
grouped_data = df_transport.groupby(['Zip Code','Model Year','Fuel','Make','Light_Duty','Vehicles'])
df_transport.groupby('Fuel').first() # Get the first entry for each month.
```

Out[9]:

	Date	Zip Code	Model Year	Make	Light_Duty	Vehicles
Fuel						
Battery Electric	10/1/2018	90000	<2006	OTHER/UNK	No	4.0
Diesel and Diesel Hybrid	10/1/2018	90000	<2006	OTHER/UNK	No	55.0
Flex-Fuel	10/14/2018	90001	2007	Type_A	Yes	78.0
Gasoline	10/1/2018	90000	2006	OTHER/UNK	Yes	1.0
Hybrid Gasoline	10/24/2018	90001	2009	OTHER/UNK	Yes	18.0
Natural Gas	10/25/2018	90001	2009	OTHER/UNK	No	2.0
Other	10/8/2018	90000	<2006	OTHER/UNK	Yes	6.0
Plug-in Hybrid	11/2/2018	90001	2012	OTHER/UNK	Yes	1.0

Another data quality issue is unwanted “string” characters in a column

Now, the intent of the “<” sign is valid, the researcher wants to show models less than 2006, but we cannot leave this “<” sign in our feature column. There are many ways to deal with this. We could create year buckets for example. In this case, we can simply remove the less than sign (given that the number of vehicles with this model year is small - this would mean removing the row and we could lose data. The strategic decisions you will need to make regarding how to handle this type of problem is beyond the scope of this introduction, but please see our readings for more resources.

Categorical Data - “Yes/No”

```
[5]: df_transport = pd.read_csv('../data/transport/untidy_vehicle_data.csv')
     df_transport.head() # Output the first five rows.
```

[5]:

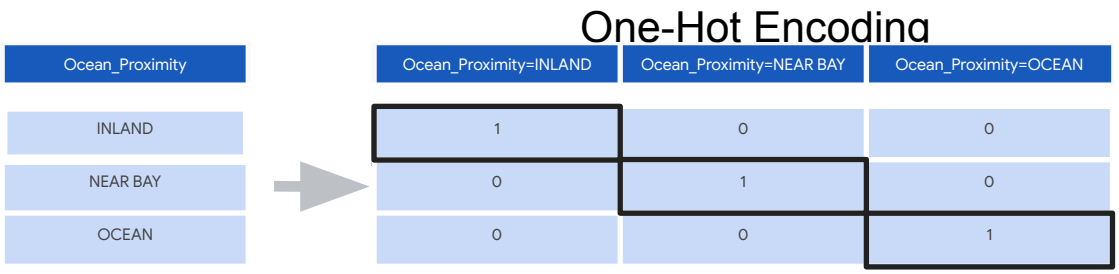
	Date	Zip Code	Model Year	Fuel	Make	Light_Duty	Vehicles
0	10/1/2018	90000	2006	Gasoline	OTHER/UNK	NaN	1.0
1	10/1/2018	NaN	2014	Gasoline	NaN	Yes	1.0
2	NaN	90000	NaN	Gasoline	OTHER/UNK	Yes	NaN
3	10/1/2018	90000	2017	Gasoline	OTHER/UNK	Yes	1.0
4	10/1/2018	90000	<2006	Diesel and Diesel Hybrid	OTHER/UNK	No	55.0

Another aspect to consider when improving data quality is to examine your categorical feature columns. In this example, we highlight the column “Light Duty”.

“Light Duty” refers to vehicle type, which can either be light duty or not or “Yes” or “No. This “Light_Duty” category has the “Yes” or “No” string values that create a categorical column.

How do we deal with this type of feature? We cannot just put words into a machine learning model.

Categorical features



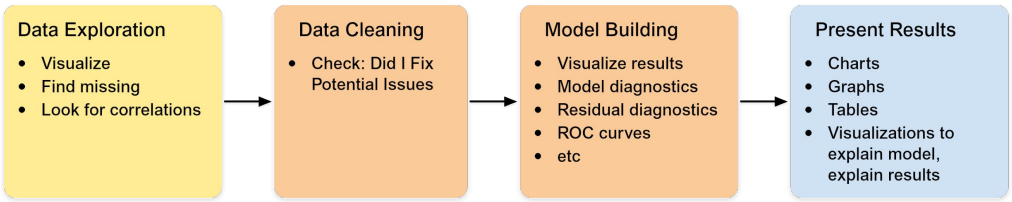
You will typically deal with categorical features by employing a process called one-hot encoding.

In one-hot encoding, you convert the categorical data into a vector of numbers. You do this because machine learning algorithms can't work with categorical data directly. Instead, you generate one boolean column for each category or class. Only one of these columns could take on the value 1 for each sample. That explains the term "one-hot encoding."

In this example, taken from the California Housing dataset used in our basic Keras lab, the feature called “ocean_proximity” contains text features. Because we can’t work with these strings directly, a boolean column for each category class is generated. Meaning, there is either a “1” or “0” in each column - you will not see a column of all zeroes and you will only see the number “1” in each column.

There are many ways to convert categorical features. Our lab will show you one way, but there are also other methods.

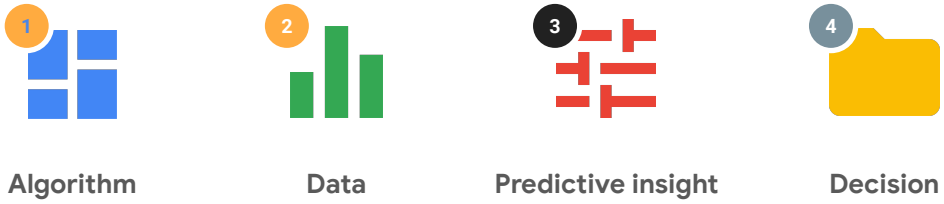
Data Quality



Improving data quality can be done before AND after data exploration. It is not uncommon to load the data and begin some descriptive analysis.

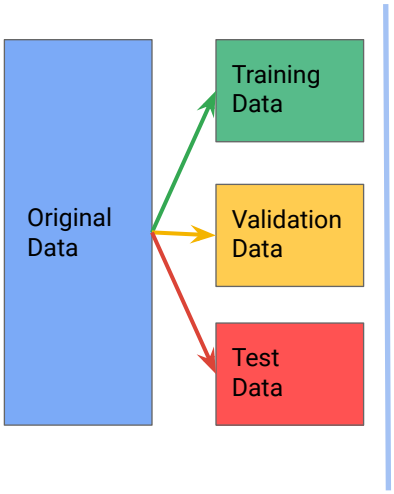
WE can “explore” and “clean” data iteratively, as you will see in the lab -- the process does not have to be a sequential process. Many times it helps to “improve the quality of our data” before we can really “explore it”!.

Machine learning is a way to use standard **algorithms** to derive **predictive insights** from **data** and make **repeated decisions**.



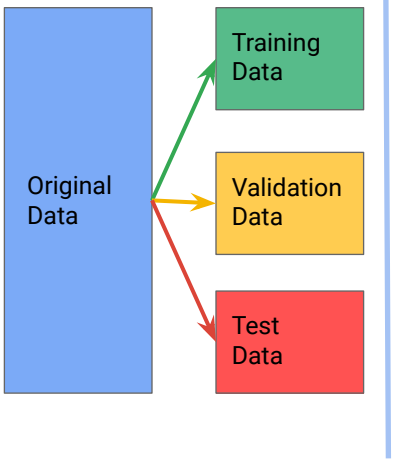
The importance of data quality cannot be overemphasized. Recall that machine learning is a way to use standard algorithms to derive predictive insights from data and make repeated decisions.

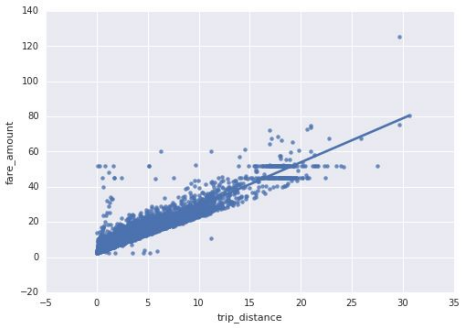
The Importance of Data Quality



Thus, in machine learning, your original source data will typically be split into a training, validation, and a test set.

The Importance of Data Quality





The quality of your source data will influence the predictive value of your model.

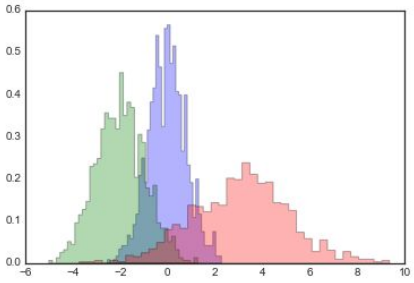
Exploratory Data Analysis

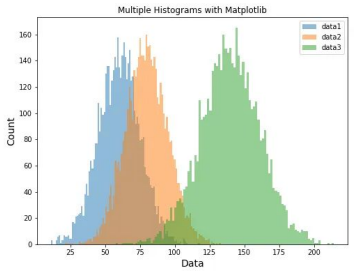
Exploratory Data Analysis

What is Exploratory Data Analysis?

How is EDA used in Machine Learning?

Data Analysis and Visualization.





In statistics, exploratory data analysis (or EDA) is an approach to analyzing data sets to summarize their main characteristics, often with visual methods. A statistical model can be used or not, but primarily EDA is for seeing what the data can tell us beyond the formal modeling or hypothesis testing task.

Exploratory data analysis (EDA) is a loosely defined term that involves using graphics and basic sample statistics (mean, median, standard deviation, etc.) to get a feeling for what information might be obtainable from your dataset. EDA is a set of techniques that allows analysts to quickly look at data for trends, outliers, and patterns. The eventual goal of EDA is to obtain theories that can later be tested in the modeling step.

Exploratory Data Analysis

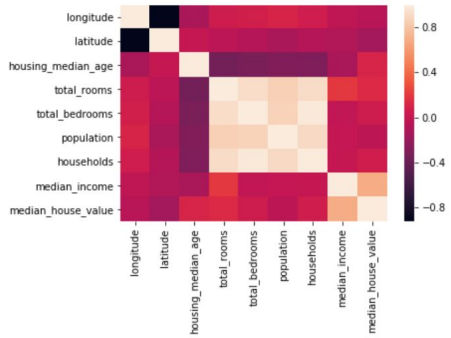
What is Exploratory Data Analysis?

How is EDA used in Machine Learning?

Data Analysis and Visualization.

```
[12]: sns.heatmap(df_USAhousing.corr())
```

```
[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f25ba7c1dd8>
```

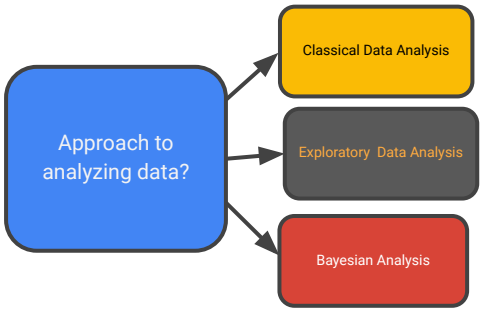


Exploratory Data Analysis (EDA) is an approach for data analysis that employs a variety of techniques (mostly graphical) to:

1. maximize insight into a data set
2. uncover underlying structure
3. extract important variables
4. detect outliers and anomalies
5. test underlying assumptions
6. develop parsimonious models
7. determine optimal factor settings.

CDA vs. EDA vs. Bayesian?

What other approaches exist and how does Exploratory Data Analysis differ from these other approaches?



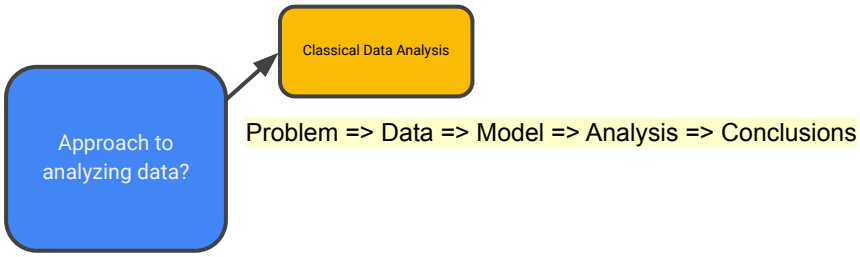
Three popular data analysis approaches are:

1. Classical
2. Exploratory (EDA)
3. Bayesian

These three approaches are similar in that they all start with a general science/engineering problem and all yield science/engineering conclusions. The difference is the sequence and focus of the intermediate steps.

CDA vs. EDA vs. Bayesian?

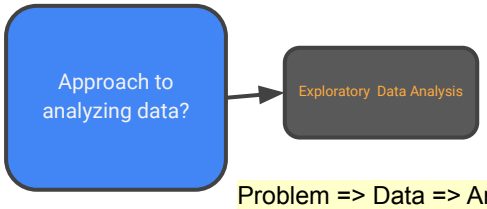
What other approaches exist and how does Exploratory Data Analysis differ from these other approaches?



For classical analysis, the data collection is followed by the imposition of a model (normality, linearity, etc.) and the analysis, estimation, and testing that follows are focused on the parameters of that model.

CDA vs. EDA vs. Bayesian?

What other approaches exist and how does Exploratory Data Analysis differ from these other approaches?



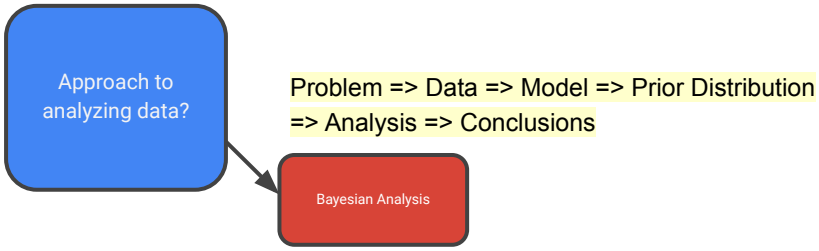
Problem => Data => Analysis => Model => Conclusions

For EDA, the data collection is not followed by a model imposition; rather it is followed immediately by analysis with a goal of inferring what model would be appropriate.

Unlike the classical approach, the Exploratory Data Analysis approach does not impose deterministic or probabilistic models on the data. On the contrary, the EDA approach allows the data to suggest admissible models that best fit the data.

CDA vs. EDA vs. Bayesian?

What other approaches exist and how does Exploratory Data Analysis differ from these other approaches?

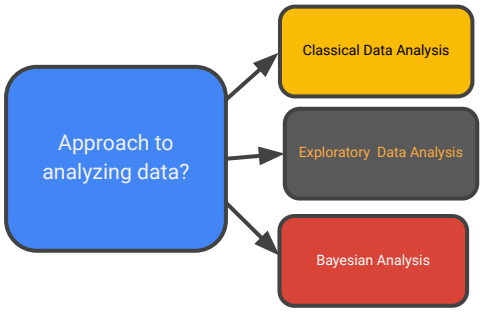


Finally, for a Bayesian analysis, the analyst attempts to answer research questions about unknown parameters using probability statements - based on prior data. They may bring their domain knowledge and / or expertise into the analysis as new information is obtained. Thus, the purpose of Bayesian analysis is to determine posterior probabilities based on prior probabilities and new information.

Posterior probability is the probability an event will happen after all evidence or background information has been taken into account. Prior probability is the probability an event will happen before you taken any new evidence into account.

CDA vs. EDA vs. Bayesian?

What other approaches exist and how does Exploratory Data Analysis differ from these other approaches?



EDA techniques are generally graphical. They include scatter plots, box plots, histograms, etc.

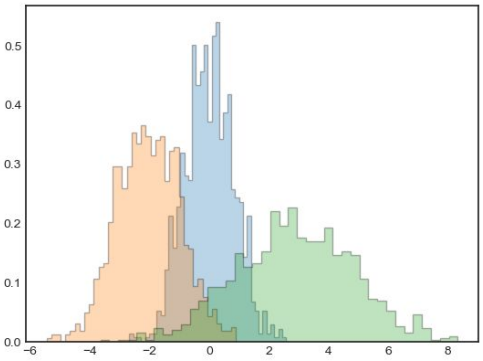
In the real world, data analysts freely mix elements of all of the above three approaches (and other approaches). The above distinctions were made to emphasize the major differences among the three approaches.

Exploratory Data Analysis

What is Exploratory Data Analysis?

How is EDA used in Machine Learning?

Data Analysis and Visualization.



How is EDA used in Machine Learning?

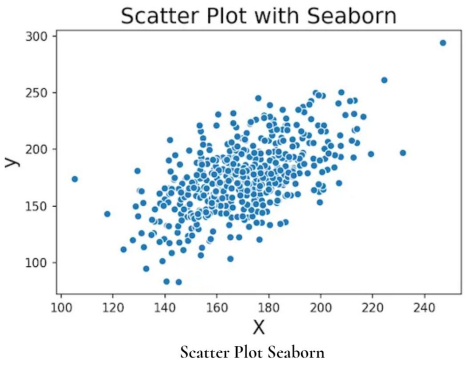
As we mentioned, the Exploratory Data Analysis approach does not impose deterministic or probabilistic models on the data. On the contrary, the EDA approach allows the data to suggest admissible models that best fit the data.

Exploratory Data Analysis

What is Exploratory Data Analysis?

How is EDA used in Machine Learning?

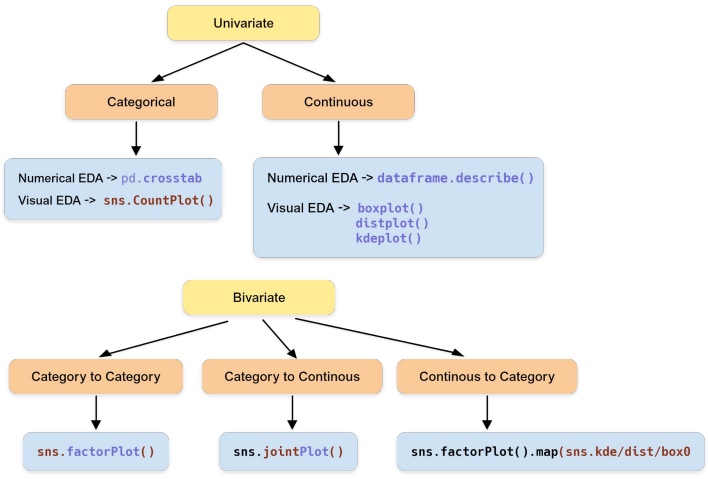
Data Analysis and Visualization.



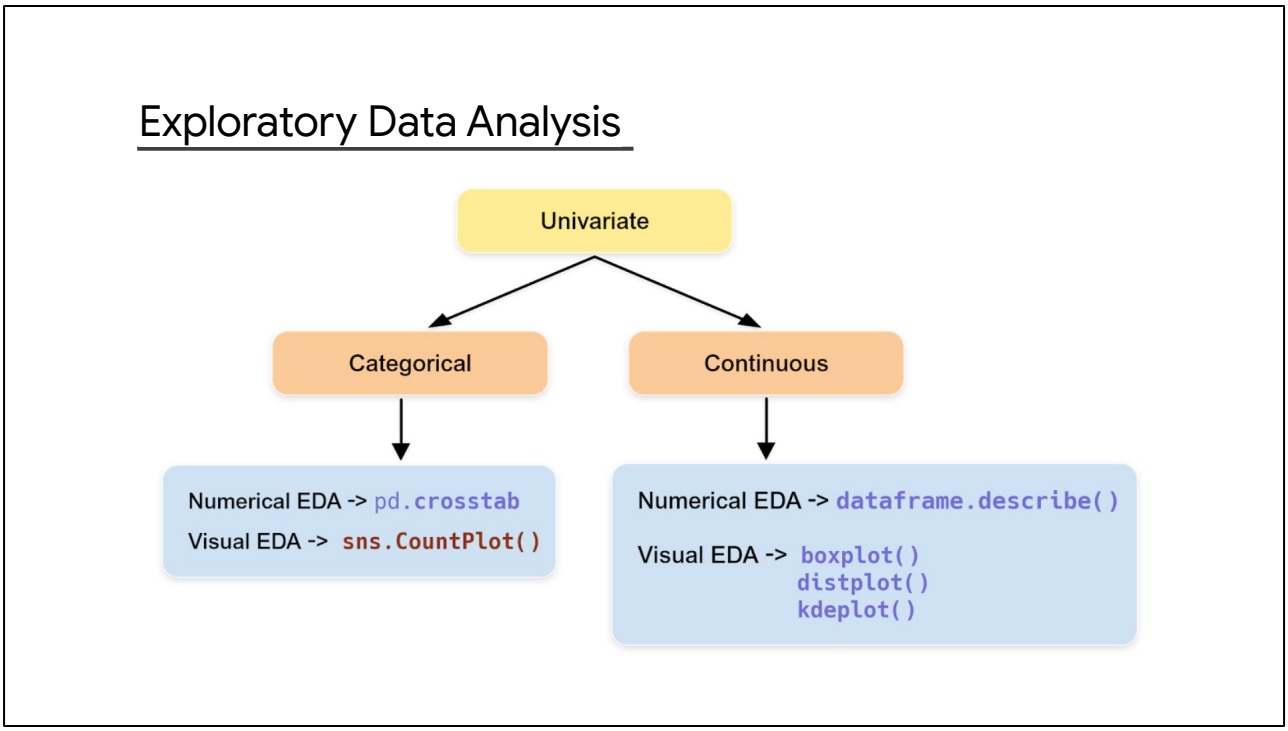
Scatter Plot Seaborn

For exploratory data analysis, the focus is on the data--its structure, outliers, and models suggested by the data.

Exploratory Data Analysis



Although there are other methods, Exploratory Data Analysis is typically performed using the following methods:



Univariate analysis is the simplest form of analyzing data. “Uni” means “one”, so in other words your data has only one variable. It doesn't deal with causes or relationships (unlike regression) and it's major purpose is to describe; It takes data, summarizes that data and finds patterns in the data.

In this example, you see two types of univariate data - categorical and continuous.

With a categorical feature type, you can perform numerical EDA using Pandas crosstab function and you can perform visual EDA using Seaborn’s countplot() function.

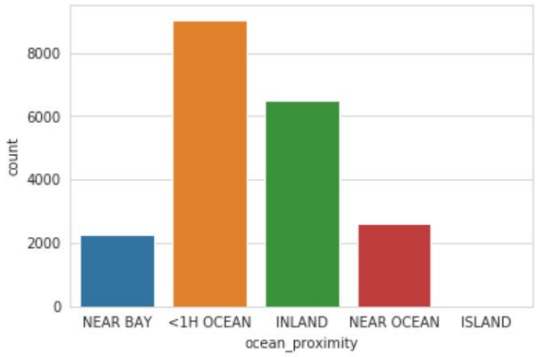
With a continuous feature type, you can perform numerical EDA using Pandas describe() function and you can visualize box plots, distribution plots, and kernel density estimation plots (or kde plots) in Python using Matplotlib or using Seaborn.

There are many EDA tools at your disposal, but that is beyond the scope of this lesson!

Univariate Data

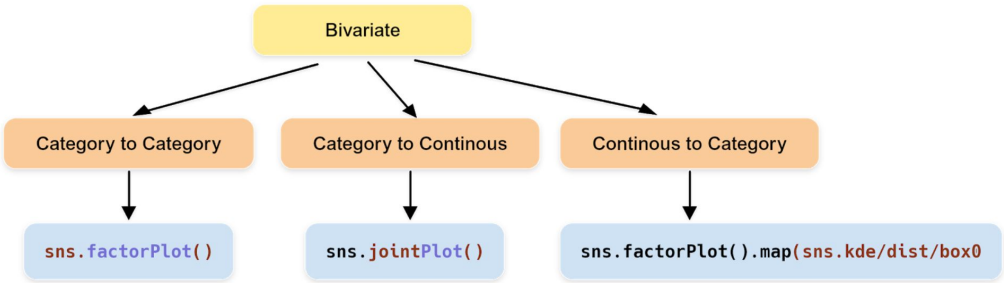
```
[17]: sns.countplot(x = 'ocean_proximity', data=df_USAhousing)
```

```
[17]: <matplotlib.axes._subplots.AxesSubplot at 0x7f25ba4ef400>
```



In this univariate data example, there is just one feature -- “ocean_proximity” -- with five categories. You can use Seaborn’s .countplot function to count the number of observations in each category. Our visualization is a simple bar chart.

Exploratory Data Analysis



Bivariate analysis means the analysis of bivariate data. It is one of the simplest forms of statistical analysis, and is used to find out if there is a relationship between two sets of values. It usually involves the variables X and Y.

We can analyze bivariate (and multivariate) data in Python using Matplotlib or using Seaborn (and there are other tools as well).

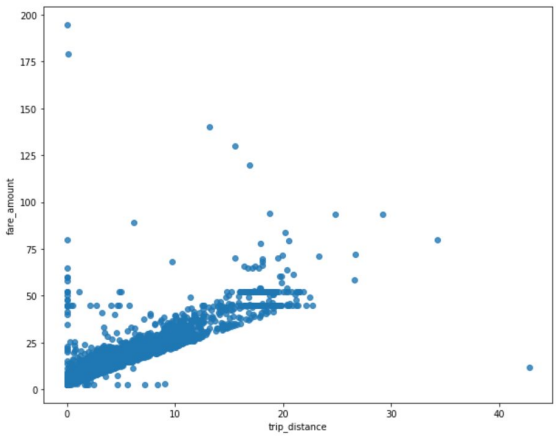
One of the most powerful features of Seaborn is the ability to easily build conditional plots; this lets us see what the data looks like when segmented by one or more variables. The easiest way to do this is through the factorplot method - which is used to draw a categorical plot onto a FacetGrid.

Seaborn's jointplot function draws a plot of two variables with bivariate and univariate graphs.

Seaborn's factorplot().map method can map a factorplot onto a kde, distribution, or boxplot chart.

Bivariate Data

```
# TODO 2
ax = sns.regplot(
    x="trip_distance", y="fare_amount",
    fit_reg=False, ci=None, truncate=True, data=trips)
ax.figure.set_size_inches(10, 8)
```

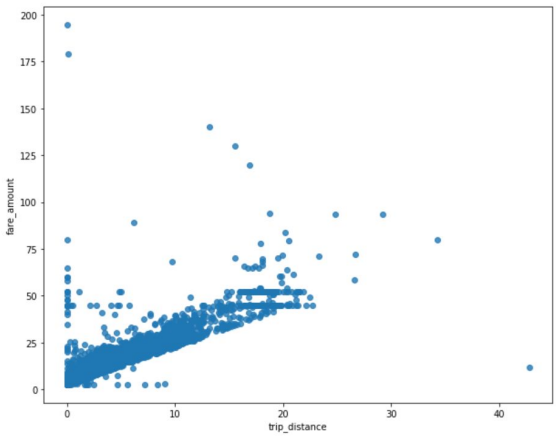


A common plot of bivariate data is the simple line plot.

In this example, we use Seaborn's sns.regplot function to visualize a linear relationship between two sets of features.

Bivariate Data

```
# TODO 2
ax = sns.regplot(
    x="trip_distance", y="fare_amount",
    fit_reg=False, ci=None, truncate=True, data=trips)
ax.figure.set_size_inches(10, 8)
```



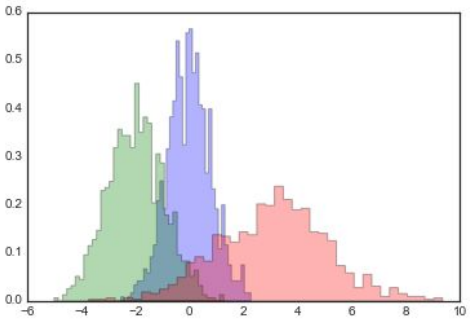
In this case trip_distance (our x label) and fare_amount (our target) appear to have a linear relationship. Note that although the majority of the data points tend to “group” together in a linear fashion, there are also outliers present as well.

Exploratory Data Analysis

What is Exploratory Data Analysis?

How is EDA used in Machine Learning?

Data Analysis and Visualization.



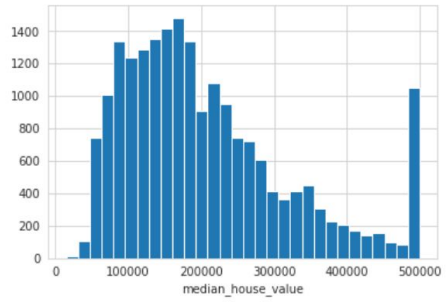
The purpose of an EDA is to find insights which will serve for data cleaning, preparation, or transformation -- which will ultimately be used in a machine learning algorithm. We use data analysis and data visualization at every step of the machine learning process where each step (Data exploration, Data cleaning, Model building, Presenting results) will belong to one notebook.

Let’s have a look at some more examples.

Histogram

```
[14]: sns.set_style('whitegrid')
      df_USAhousing['median_house_value'].hist(bins=30)
      plt.xlabel('median_house_value')
```

```
[14]: Text(0.5, 0, 'median_house_value')
```



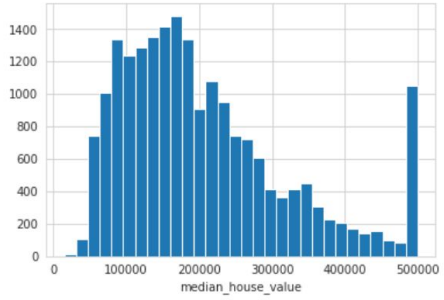
A histogram is a graphical display of data using bars of different heights.

In a histogram, each bar groups numbers into ranges. Taller bars show that more data falls in that range. A histogram displays the shape and spread of continuous sample data.

Histogram

```
[14]: sns.set_style('whitegrid')
      df_USAhousing['median_house_value'].hist(bins=30)
      plt.xlabel('median_house_value')
```

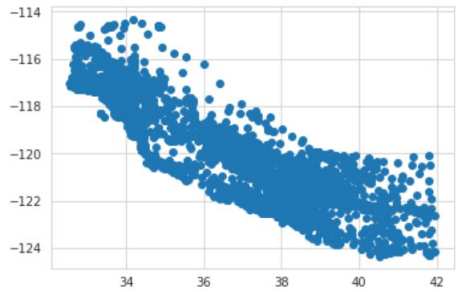
```
[14]: Text(0.5, 0, 'median_house_value')
```



In this example, we use Seaborn's distplot function to plot a histogram of the feature "median house value".

Scatter Plot

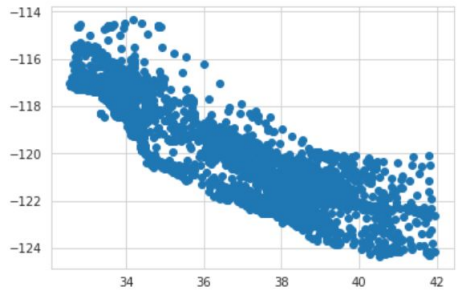
```
x = df_USAhousing['latitude']
y = df_USAhousing['longitude']

plt.scatter(x, y)
plt.show()
```



Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

Scatter Plot

```
x = df_USAhousing['latitude']
y = df_USAhousing['longitude']

plt.scatter(x, y)
plt.show()
```



In this example, we use Matplotlib's plyplot function to plot a scatter plot.

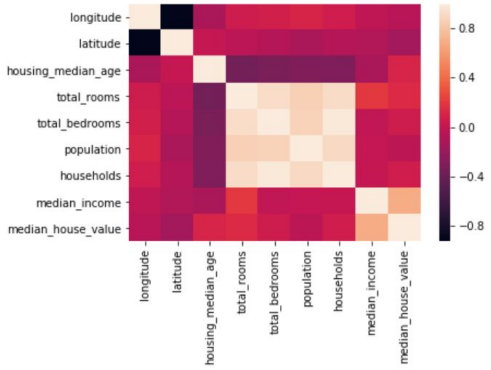
A scatter plot is a graph in which the values of two variables are plotted along two axes - the pattern of the resulting points revealing any correlation present.

Here, we can see that by plotting housing location latitude on the "x" axis and "longitude" on the "y" axis, we see that the resulting revealed correlation pattern is the state of California.

Correlations

```
[12]: sns.heatmap(df_USAhousing.corr())
```

```
[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f25ba7c1dd8>
```

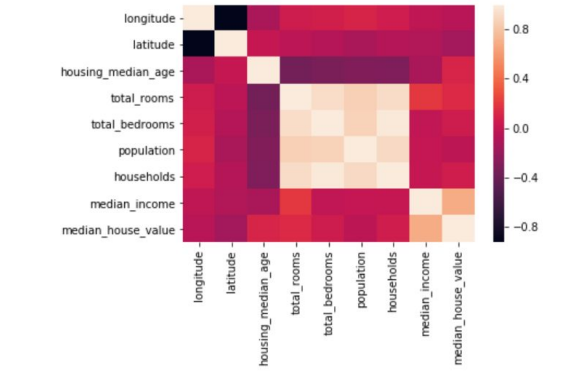


In this example, we use Seaborn’s

Correlations - Multivariate

```
[12]: sns.heatmap(df_USAhousing.corr())
```

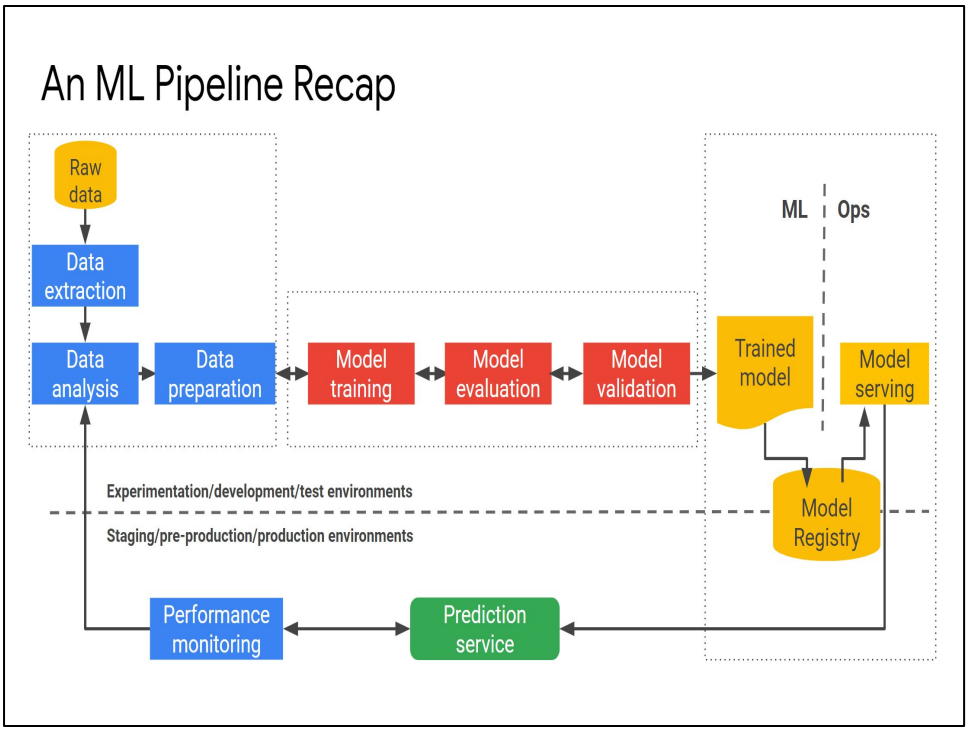
```
[12]: <matplotlib.axes._subplots.AxesSubplot at 0x7f25ba7c1dd8>
```



heatmap function to show correlations. A heatmap is a graphical representation of data that uses a system of color-coding to represent different values.

For example, you can see the correlation between all the features in your dataset. The lighter the shade, the stronger the correlation. This is a quick and easy way to see which features may influence your target.

If you think about it, a heatmap plots multiple variables, and can be thought of as an example of multivariate graphical analysis, another area of Exploratory Data Analysis.



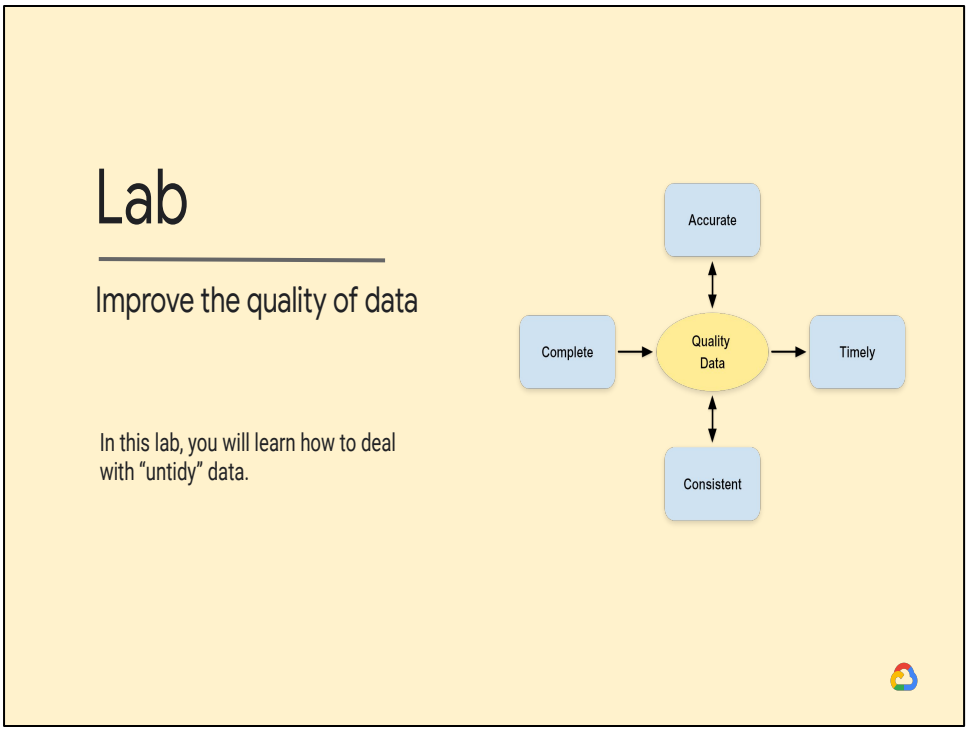
So, to summarize, data analysis, which is the second step in the ML pipeline, is a crucial milestone and must be used to prepare the data before model training.

The purpose of exploratory data analysis includes being able to:

- Gain maximum insight into the data set and its underlying structure.
- Create a list of outliers or other anomalies.

And Identify the most influential features.

There are many more ways to explore, analyze, and plot data -- make it a goal to expand your knowledge of them. Have fun!

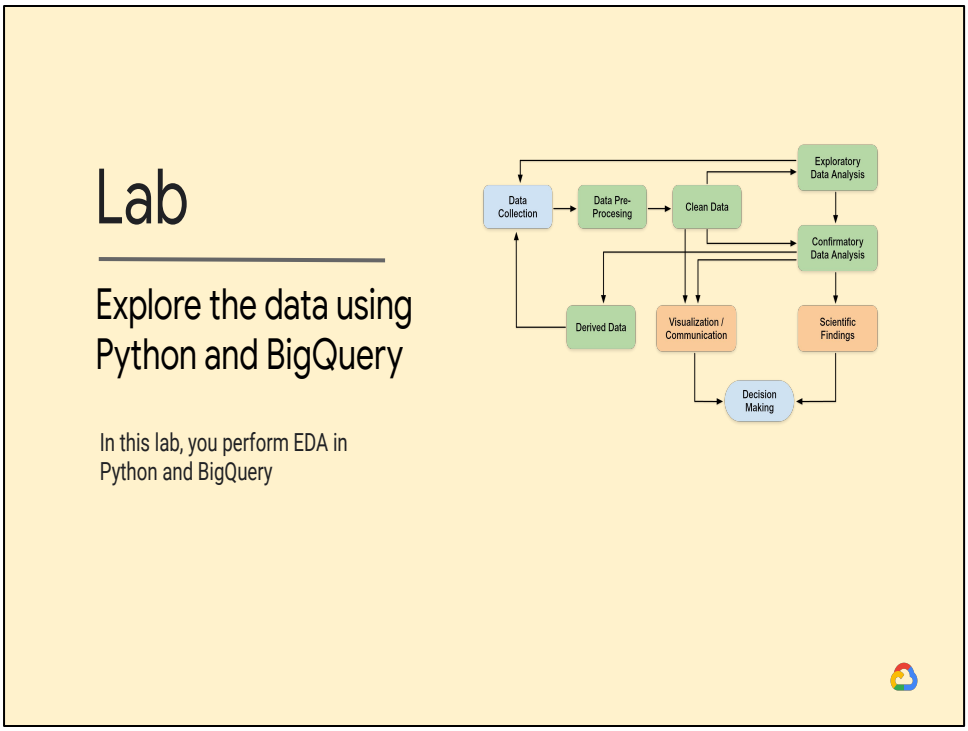


This lab focuses on improving data quality.

Recall that machine learning models can only consume numeric data, and that numeric data should be "1"s or "0"s. Data is said to be "messy" or "untidy" if it is missing attribute values, contains noise or outliers, has duplicates, wrong data, upper/lower case column names, and is essentially not ready for ingestion by a machine learning algorithm.

This lab presents and solves some of the most common issues of "untidy" data. Note that different problems will require different methods, and they are beyond the scope of this lab.

Link to lab: [ML on GCP C2] Improving Data Quality (qwiklabs). git repo link here.

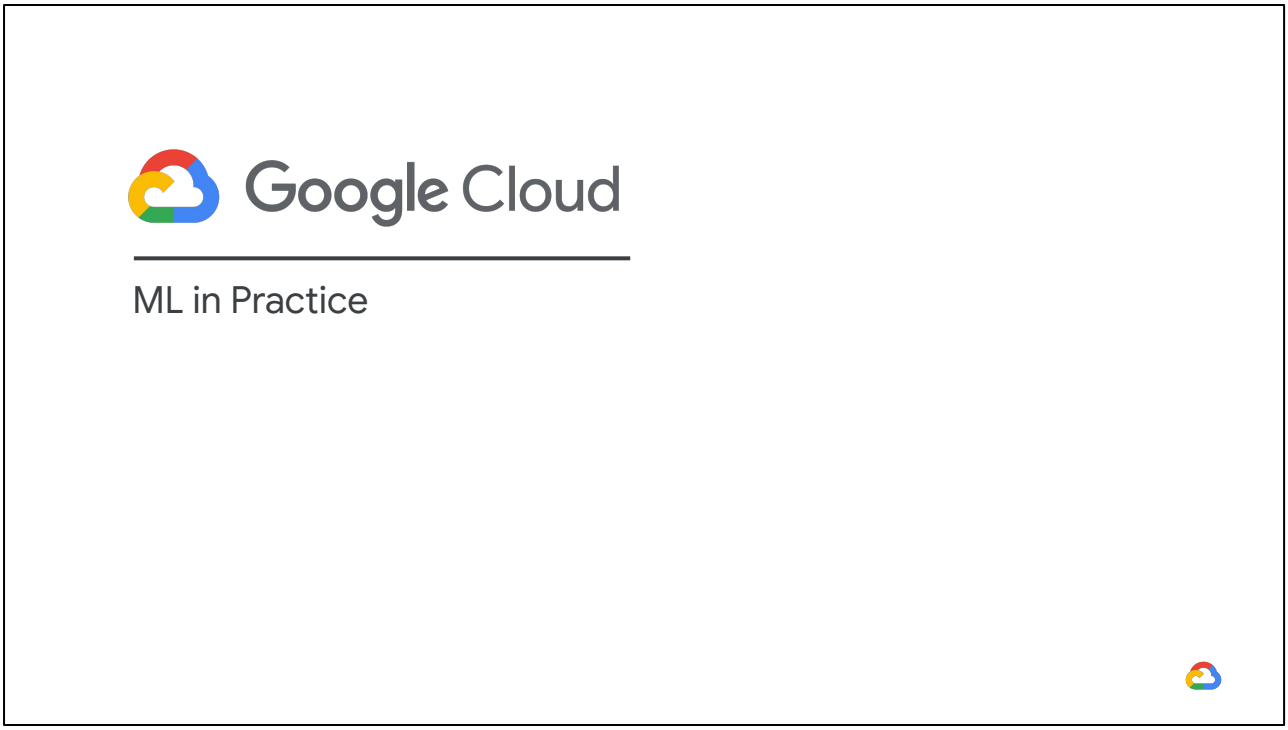


In this lab, you will perform exploratory data analysis on the USA Housing dataset and the New York Taxi Fare Dataset as well.

You will create Seaborn plots for Exploratory Data Analysis in Python and Big Query.

First, you'll learn how to analyze a Pandas Dataframe.
Next, you'll learn how to create Seaborn plots for Exploratory Data Analysis in Python.
Next, you'll learn how to write a SQL query to pick up specific fields from a BigQuery dataset.
Then, you will learn about Exploratory Analysis in BigQuery.

Link to lab: [ML on GCP C2] Exploratory Data Analysis Using Python and BigQuery (qwiklabs). git repo link here.

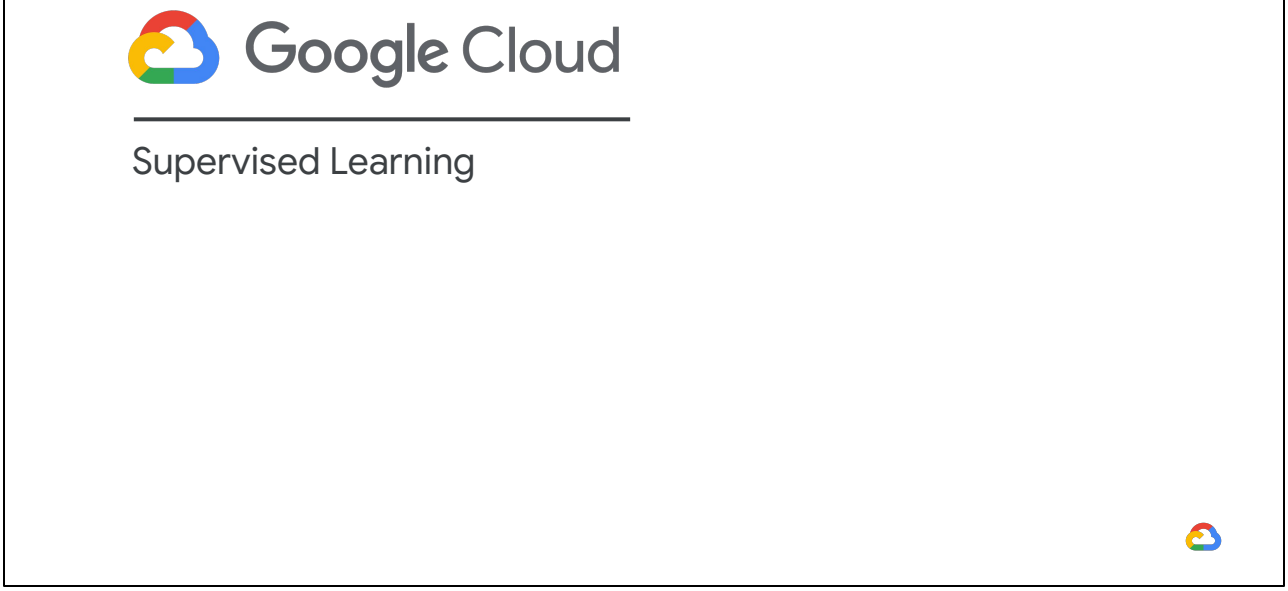


ML in Practice

ML in Practice

You will start out talking about the historical evolution of machine learning, from its use in applications like astronomy to now where it is used widely in commercial applications to automate many tasks and augment the way those applications work. For example, machine learning is used to read house numbers from Street View images to add labels in Google Maps. While talking about the historical evolution, you'll also describe how deep learning techniques incorporate many of the improvements brought on by earlier machine learning methods like decision trees and random forests.

https://pixabay.com/en/earth-north-star-north-navigation-23592/ (cc0)
https://www.google.com/intl/en/about/main/machine-learning-qa/
https://drive.google.com/a/google.com/file/d/0B32iIRwWYc2namRTYXRCa1I3TjQ/edit?usp=sharing



We discussed ML as a process and how Google has adopted several philosophical positions that have been crucial to our ML success. What you haven't done yet is dive into what ML is and how it works. That's what you'll do now.

In the next few slides we'll cover:

- Supervised learning, which is one branch of machine learning where you give the model labeled examples of what it should learn.
- A history of ML, to survey the algorithms of the last 50 years, and to understand why neural networks are so prominent at this moment.

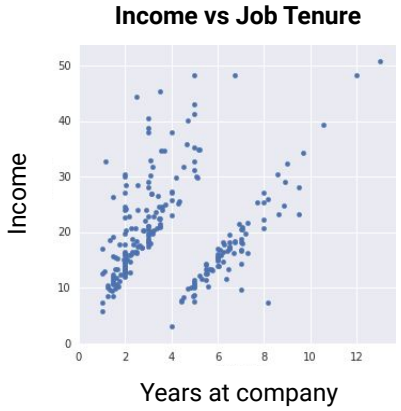
Let's start with Supervised Machine Learning.

Unsupervised and supervised learning are the two types of ML algorithms

Example Model: Clustering

Is this employee on the "fast-track" or not?

In unsupervised learning, data is not labeled.

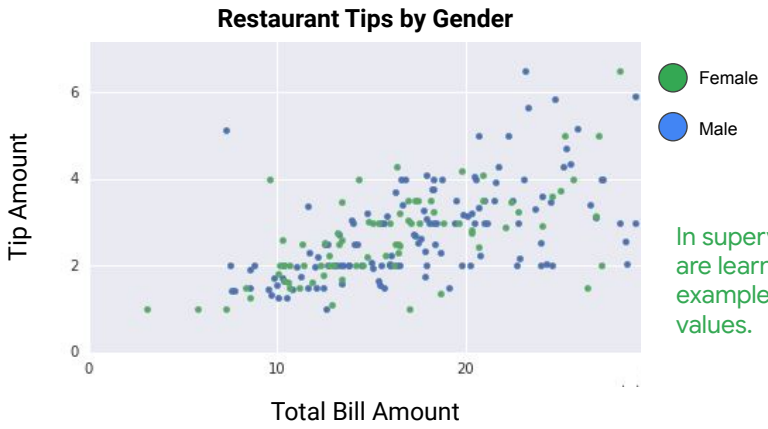


Two of the most common classes of machine learning models are supervised and *unsupervised* ML models. The key difference is that with supervised models, we have labels, or in other words, the correct answers to whatever it is that we want to learn to predict.

In unsupervised learning, the data does not have labels.

This graph is an example of the sort of problem that an unsupervised model might try to solve. Here, you want to look at tenure and income, and then group or cluster employees, to see whether someone is on the fast track. Critically, there is no ground truth here; management doesn't, as far as you know, have a big table of people they are going to promote fast, and those they are not going to promote. Consequently, unsupervised problems are all about discovery, about looking at the raw data, and seeing if it naturally falls into groups. At first look, it seems that there are two distinct clusters or groups that I could separate nicely with a line.

Supervised learning implies the data is already labeled



In supervised learning we are learning from past examples to predict future values.

Ryan: In this chapter though, we'll be focused on supervised machine learning problems, like this one. The critical difference is that with supervised learning, we have some notion of a "label," or one characteristic of each data point that we care about a lot.

Typically, this is something you know about in historical data, but you don't know in real time. You know other things, which you call predictors, and you want to use those predictors to predict the thing you don't know.

For example, let's say you are the waiter in a restaurant. You have historical data of the bill amount and how much different people tipped. Now, you are looking at the group sitting at the corner table. You know what their total bill is, but, you don't know what their tip is going to be. In the historical data, the tip is a label. You create a model to predict the tip from the bill amount. Then, you try to predict the tip, in real time, based on the historical data and the values that you know for the specific table.

Regression and classification are supervised ML model types

	total_bill	tip	sex	smoker	day	time
2	16.99	1.01	Female	No	Sun	Dinner
3	10.34	1.66	Male	No	Sun	Dinner
4	21.01	3.5	Male	No	Sun	Dinner
5	23.68	3.31	Male	No	Sun	Dinner
6	24.59	3.61	Female	No	Sun	Dinner
7	25.29	4.71	Male	No	Sun	Dinner
8	8.77	2	Male	No	Sun	Dinner
9	26.88	3.12	Male	No	Sun	Dinner

Option 1
Regression Model
Predict the tip amount

Option 2
Classification Model
Predict the sex of the customer

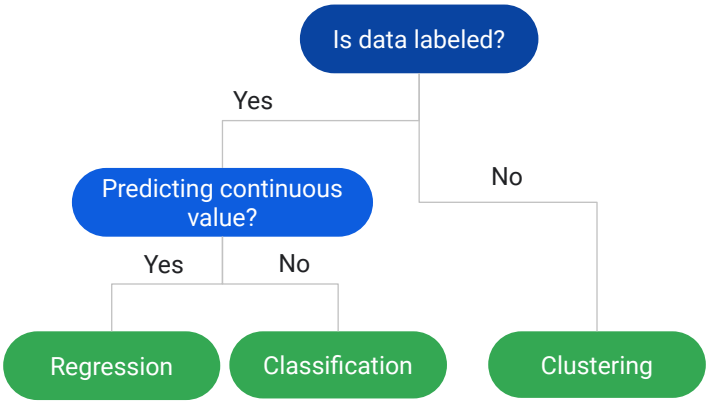
Ryan: Within supervised ML there are two types of problems: regression and classification. To explain them, let's dive a little deeper into this data.

In this dataset of tips, an example dataset that comes with the Python package *seaborn*, each row has many characteristics, such as total bill, tip, and sex. In machine learning, we call each row an "example." You'll choose one of the columns as the characteristic we want to predict, called the "label," and you'll choose a set of the other columns, which are called the "features."

In model option 1, you want to predict the tip amount, therefore the column *tip* is your label. You can use one, all, or any number of the other columns as my features to predict the tip. This will be a regression model because *tip* is a continuous label.

In model option 2, you want to predict the sex of the customer, therefore the column *sex* is the label. Once again, you will use some set of the rest of the columns as your features, to try and predict the customer's sex. This will be a classification model because our label *sex* has a discrete number of values or classes.

The type of ML problem depends on whether or not you have labeled data and what you are interested in predicting



In summary, depending on the problem you are trying to solve, the data you have, explainability, etc. will determine which machine learning methods you use to find a solution.

Your data isn't labeled? You won't be able to use supervised learning then, and will have to resort to clustering algorithms to discover interesting properties of the data.

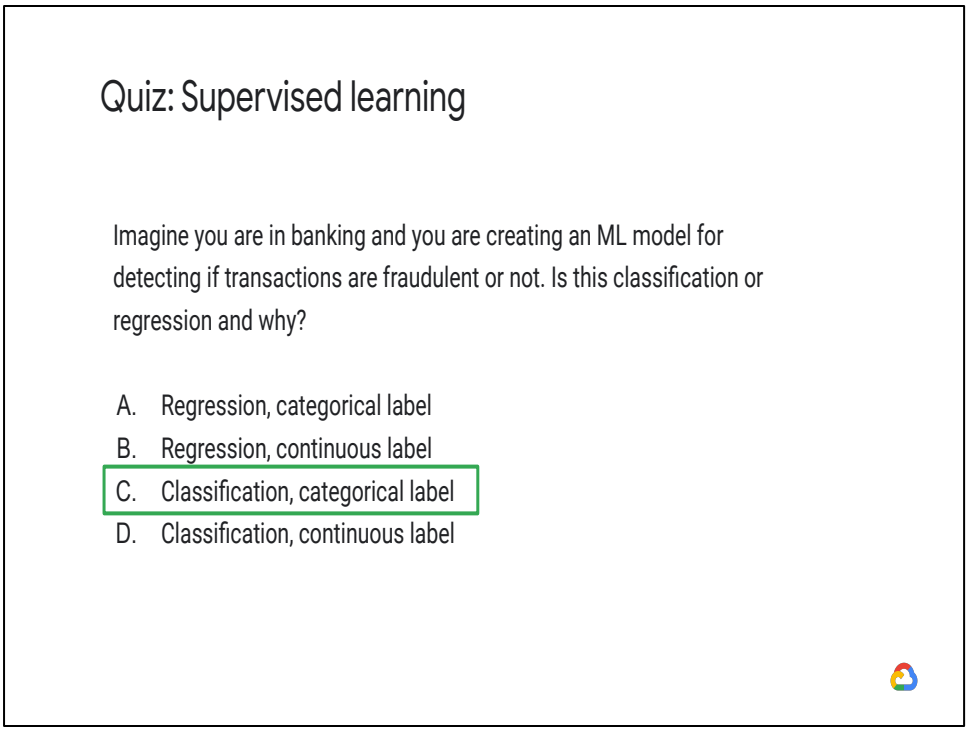
Your data is labeled and the label is dog breed, which is a discrete quantity since there are a finite number of dog breeds? You should use a classification algorithm. If instead the label is dog weight, which is a continuous quantity, you should use a regression algorithm. The label, again, is the thing that you are trying to predict. In supervised learning, you have some data with the correct answers.

Quiz: Supervised learning

Imagine you are in banking and you are creating an ML model for detecting if transactions are fraudulent or not. Is this classification or regression and why?

A. Regression, categorical label
B. Regression, continuous label
C. Classification, categorical label
D. Classification, continuous label

Question: Imagine you are in banking and you are creating an ML model for detecting if transactions are fraudulent or not. Is this classification or regression and why?

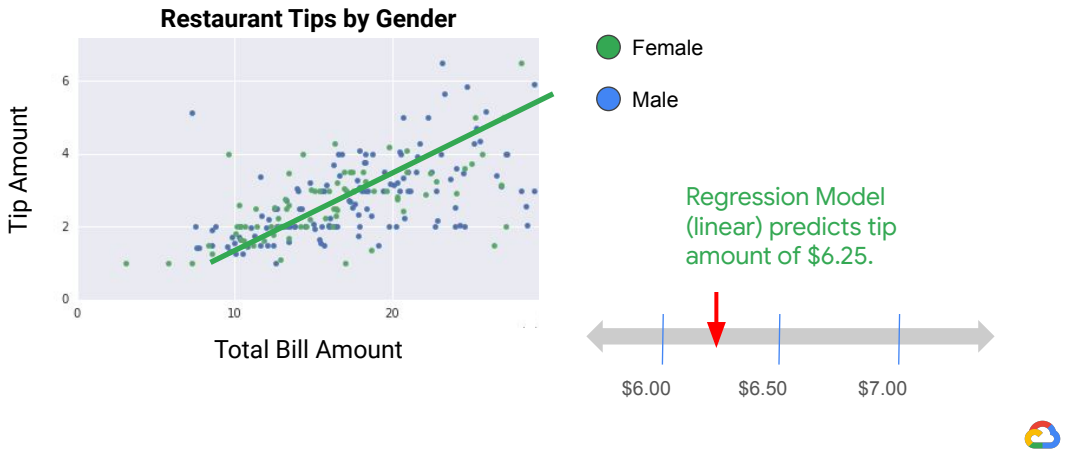


Answer: The correct answer is classification, categorical label. This is a binary classification problem because there are two possible classes for each transaction: fraudulent or not fraudulent. In practice, you may actually have a third class, uncertain. This way depending on your classification threshold it could send any cases that it can't firmly place into the fraudulent or not fraudulent buckets to a human to have a closer look. It is often good practice to have a human in the loop when performing machine learning.

You can eliminate Regression, categorical label and Classification, continuous label because the model types have the opposite label type than they should.

Regression, continuous label at least is a correct pairing however it is incorrect because this is a classification problem so you would not use regression. You could, also create a regression model such as predicting the number of fraudulent transactions, fraudulent transaction amounts, etc.

Use regression for predicting continuous label values



We looked at the tips dataset and said that we could use either the tip amount as the label or the sex of the customer as the label.

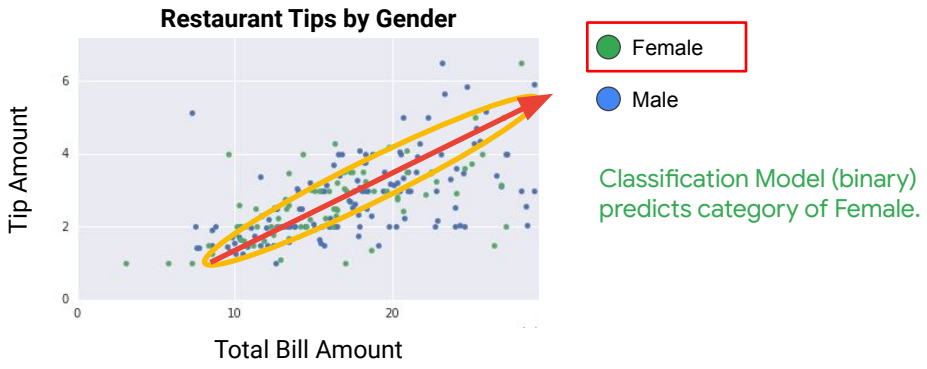
In option 1, we are treating the tip amount as the label and want to predict it, given the other features in the dataset. Let's assume that you are using only one feature -- just the total bill amount -- to predict the tip. Because tip is a continuous number, this is a regression problem. In regression problems, the goal is to use mathematical functions of different combinations of our features to predict the continuous value of our label. This is shown by this line where for a given total bill amount times the slope of the line we get a continuous value for tip amount. Perhaps the average tip rate is 18% of the total bill. Then, the slope of the line would be 0.18.
And by multiplying the bill amount by 0.18, we'll get the predicted tip.

This linear regression with only one feature generalizes to additional features. In that case, we have a multi-dimensional problem, but the concept is the same. The value of each feature for each example is multiplied by the gradient of a hyperplane, which is just the generalization of a line, to get a continuous value for the label.

In regression problems, we want to minimize the error between our predicted continuous value and the label's continuous value, usually using mean squared

error.

Use classification for predicting categorical label values



In Option 2, we are going to treat sex as our label and predict the gender of the customer using data from the tip and total bill. Of course, as you can see from the data, this is a bad idea -- the data for men and women is not really separate -- and we would get a terrible model if we did this. But trying to do this helps me illustrate what happens when the thing you want to predict is categorical and not continuous.

The values that the sex column takes, at least in this dataset, are discrete (male or female). Because sex is categorical and we are using the sex column of the dataset as our label, the problem is a classification problem.

In classification problems, instead of trying to predict a continuous variable, we are trying to create a decision boundary that separates the different classes.
So, in this case, there are two classes of sex; Female and Male. A linear decision boundary would form a line (or a hyperplane in higher dimensions) with each class on either side.
For example, we might say that if the tip amount is > 0.18 * total bill amount, then we predict that the person making the payment was male.
This is shown by the red line.
But that doesn't work very well for this dataset.

Men seem to have higher variability while women tend to tip in a more narrow band. This is an example of a non-linear decision boundary, shown by the yellow ellipse in the graph.

How do we know the red decision boundary is bad, and the yellow decision boundary is better? In classification problems, we want to minimize the error or misclassification between our predicted class and the label's class. This is done usually using cross entropy.

Even if we are predicting the tip amount, perhaps we don't need to know the exact tip amount. Instead, we want to determine whether the tip amount will be high, average, or low. We could define a high tip amount as > 25%, average tip amount as between 15% and 25%, and a low tip amount as being below 15%. In other words, we could discretize the tip amount. And now, predicting the tip amount or more appropriately the tip class becomes a classification problem.

In general, a raw continuous feature can be discretized into a categorical feature.

Later in the specialization, we will talk about the reverse process -- a categorical feature can be "embedded" into a continuous space.
It really depends on the exact problem you are trying to solve and what works best. Machine learning is all about experimentation.

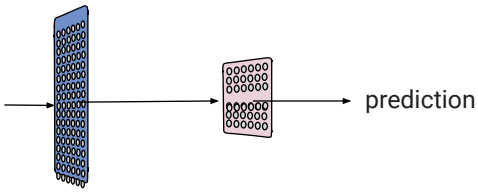
Both of these problem types, regression and classification, can be thought of as "prediction" problems, in contrast to unsupervised problems, which are more like "description" problems.

A data warehouse can be a source of structured data training examples for your ML model

```
SELECT
  gestation_weeks,
  mother_age,
  cigarette_use,
  alcohol_use,
  weight_gain_pounds
FROM
  `bigquery-public-data.samples.natality`
WHERE cigarette_use is not null AND alcohc
```

weight	year	mother_age	gestation_weeks	cigarette_use	alcohol_use
7.86	2003	25	39	false	false
7.5	2003	21	39	false	false
8.06	2004	29	40	false	false
7.56	2004	38	37	false	false
7.06	2003	22	38	false	false

Data on births is sourced from our BigQuery Data Warehouse using SQL.



Now, where does this data come from? The tips dataset is what we call structured data -- consisting of rows and columns -- and a very common source of structured data for machine learning is your data warehouse. Unstructured data is things like pictures, audio, or video.

Here is showing you a natality dataset, a public dataset of medical information. It is a public dataset in BigQuery, and you will use it later in the specialization but for now assume that this dataset is in your data warehouse

Let's say you want to predict the gestation weeks of the baby. In other words, you want to predict when the baby is going to be born.

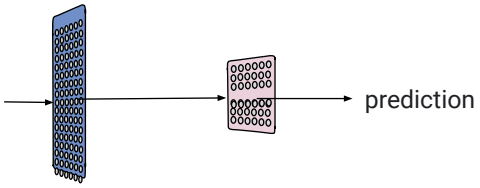
You can do a SQL SELECT statement in BigQuery to create a ML dataset -- you will choose input features to the model, things like mother's age and weight gain in pounds, and the label, gestation weeks.

Because gestation weeks is a continuous number, this is a regression problem.

Making predictions from structured data is very commonplace, and that is what you focus on in the first part of the specialization.

Since baby weight is a continuous value, use regression to predict

weight	year	mother_age	gestation_weeks	cigarette_use	alcohol_use
7.86	2003	25	39	false	false
7.5	2003	21	39	false	false
8.06	2004	29	40	false	false
7.56	2004	38	37	false	false
7.06	2003	22	38	false	false



Weight is stored as a floating point number, representing a continuous (real) value.

Regression DNN Model

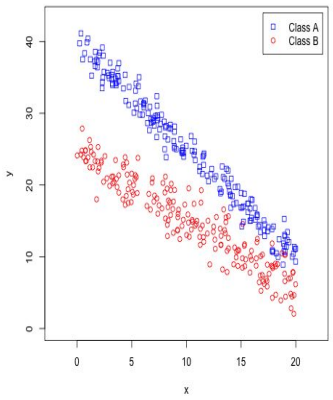
Of course, this medical dataset can be used to predict other things too. Perhaps you want to predict baby weight using the other attributes as our features.
Baby weight can be an indicator of health: when a baby is predicted to have a low birth weight, the hospital will usually have equipment such as an incubator handy. So, it can be important to be able to predict a baby's weight.

The label here would be *baby weight,* and it is a continuous variable. It's stored as a floating point number, which would make this a regression problem.

Quiz: Regression/Classification

Is this dataset a good candidate for linear regression and/or linear classification?

A. Linear classification
B. Both
C. None of the above

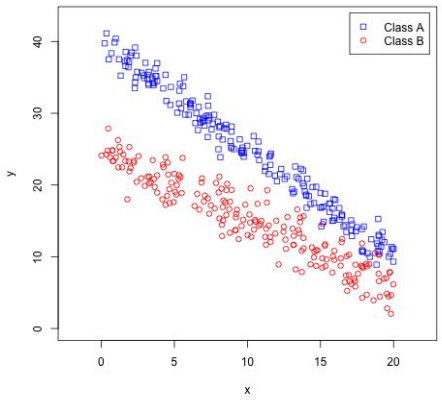


Question: Is this dataset a good candidate for linear regression and/or linear classification?

Quiz: Regression/Classification

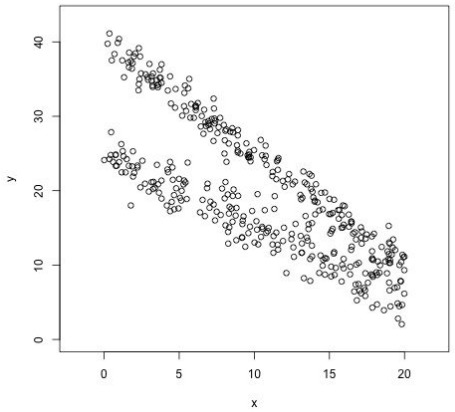
Is this dataset a good candidate for linear regression and/or linear classification?

A. Linear classification
B. Both
C. None of the above



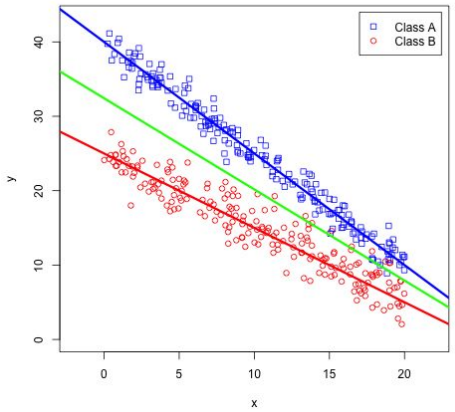
Answer: The correct answer is both. Let's investigate why.

Is this dataset a good candidate for linear regression and/or linear classification?



Let’s step back and look at the dataset with both classes mixed. Without the different colors and shapes to aid us, the data appears to be one noisy line with a negative slope and positive intercept. Since it appears quite linear this will most likely be a good candidate for linear regression, where what you are trying to predict is the value for y.

Is this dataset a good candidate for linear regression and/or linear classification?



Adding the different colors and shapes back in, it is much more evident that this dataset is actually two linear series with some gaussian noise added. The lines have slightly different slopes and different intercepts, and the noise has different standard deviations. The lines have been plotted here to show you this is most definitely a linear dataset by design albeit a bit noisy. This would be a good candidate for linear regression. Despite there being two distinct linear series, let's first look at the results of a one dimensional linear regression predicting y from x to start building an intuition. Then you'll see if you can do better!

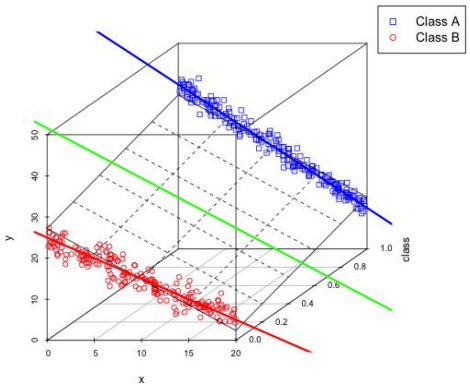
The green line here is the fitted linear equation from linear regression. Notice that it is far away from each individual class' distribution because class B pulls the line away from class A and vice versa. It ends up approximately bisecting the space between the two distributions. This makes sense since with regression we optimize our loss of mean squared error so with an equal pull from each class the linear regression should have the lowest mean squared error in between the two classes approximately equidistant from their means.

Since each class is a different linear series with different slopes and intercepts, we would actually have much better accuracy by performing a linear regression for each class which should fit very closely to each of the lines plotted here. Even better, instead of performing a one dimensional linear regression predicting the value of y from one feature, x, we could perform a two dimensional linear regression predicting y from two features: x and the class of the point. The class feature could be a 1 if the point belongs to class A and a 0 if the point belongs to class B. Instead of a line, it

would form a 2D hyperplane.

Let’s see how that would look.

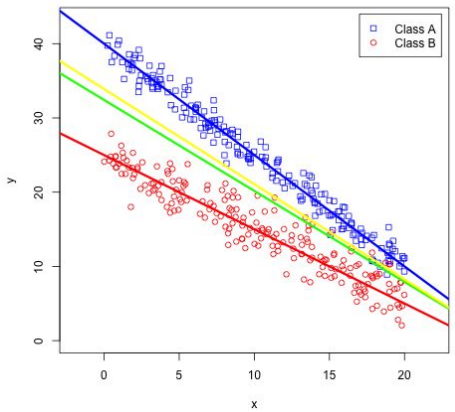
Is this dataset a good candidate for linear regression and/or linear classification?



Here are the results of the 2D linear regression. To predict our label y, we used two features: x and class. As you can see a 2D hyperplane has been formed between the two sets of data, which are now separated by the class dimension. I've also included the true lines for both class A and class B as well as the 1D linear regression's line of best fit. The plane doesn't completely contain any of the lines, due to the noise of the data tilting the two slopes of the plane. Otherwise with no noise, all three lines would be perfectly on the plane.

Also, we have kind of already answered the other portion of the quiz question about linear classification, because the linear regression line does a really great job already of separating the classes. So this looks like a very good candidate for linear classification as well. But would it produce a decision boundary exactly on the 1D linear regression's line of best fit? Let's find out!

Is this dataset a good candidate for linear regression and/or linear classification?

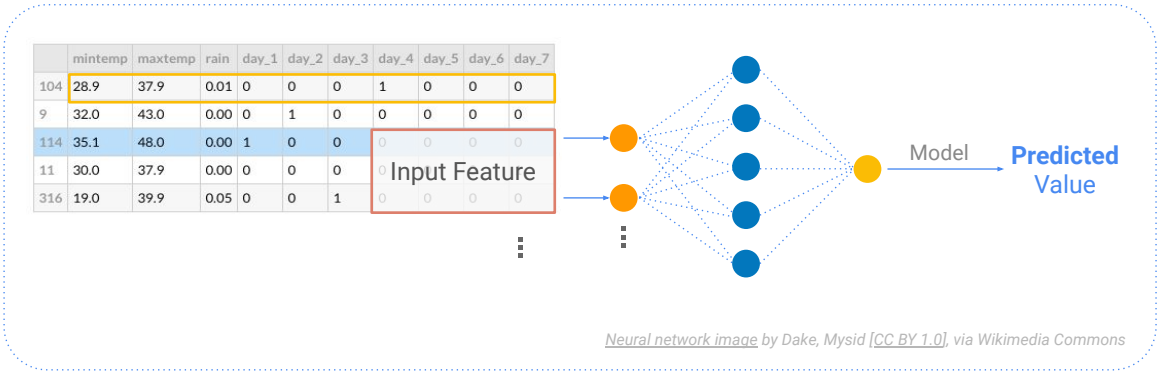


Plotted in yellow is the output of a one dimensional linear classifier, logistic regression. Notice that it is very close to the linear regression's green line but not exactly. Why could this be? Remember, I mentioned that regression models usually use mean squared error as their loss function whereas classification models tend to use cross entropy. So what is the difference between the two? Without going into the details too much just yet, there is a quadratic penalty for mean squared error so it is essentially trying to minimize the euclidean distance between the actual label and predicted label.

On the other hand, with classification's cross entropy the penalty is almost linear looking when the predicted probability is close to the actual label but as it gets farther away it becomes exponential when it gets close to predicting the opposite class of the label. Therefore, if you look closely at the plot, the most likely reason the classification decision boundary line has a slightly more negative slope is so that some of those noisy red points, red being the noiser distribution, fall on the other side of the decision boundary and lose their high error contribution. Since they are close to the line their error contribution would be small for linear regression because not only is it the error only quadratic but there is no preference to be on one side of the line or the other for regression, as long as the distance is as small as possible.

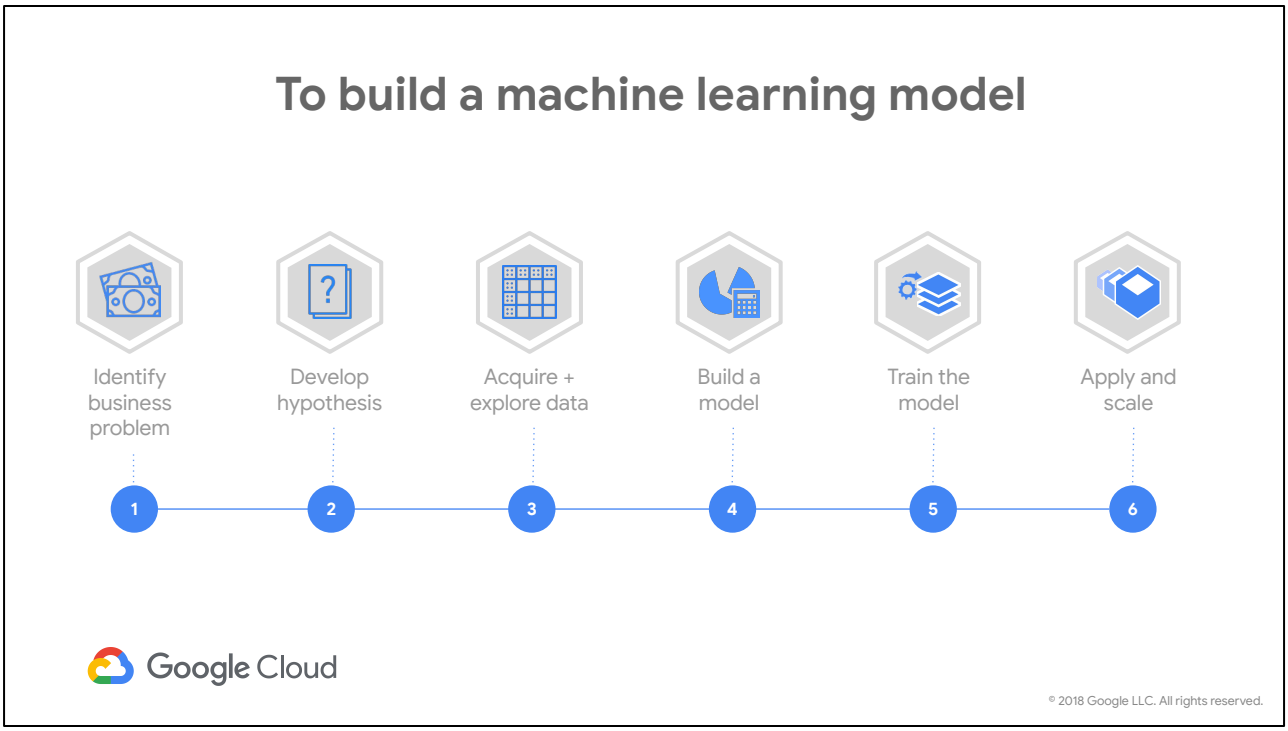
So as you can see, this dataset is a great fit for both linear regression and linear classification, unlike when we looked at the tips dataset where it was only acceptable for linear regression and would be better for nonlinear classification.

The point of ML is to make predictions

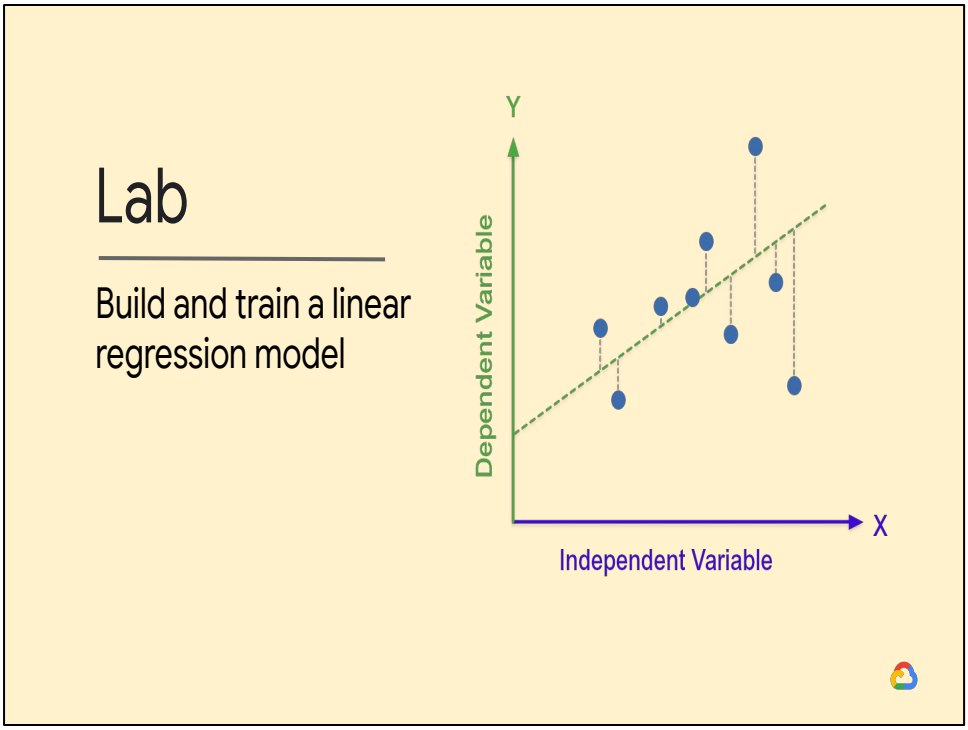


Google Cloud

Neural network image by Dake, Mysid [CC BY 1.0], via Wikimedia Commons



Specifically you must collect information about your use case. Build a model to make sense of it. Train that model repeatedly and at scale with more data and then apply that model into a solution to make it useful.



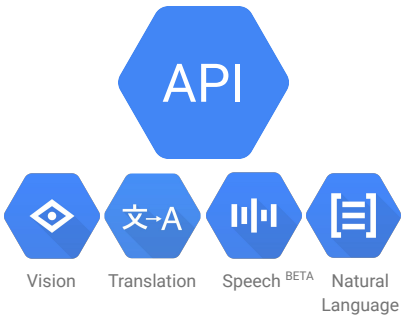
In this lab, you will build and train a linear regression model.You will first load data into a Pandas dataframe and analyze it.You will also create seaborn plots to perform exploratory data analysis.Then, you will train a linear regression model using SciKit Learn.

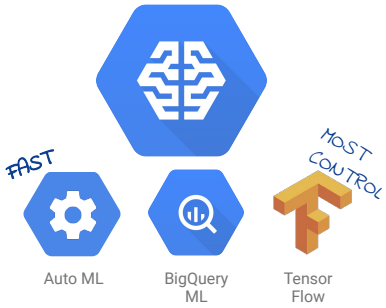
Link to lab: [ML on GCP C2] Introduction to Linear Regression. github repo here.

Google Cloud

BigQuery Machine Learning

BigQuery ML is a way to build custom models





Pre-trained models

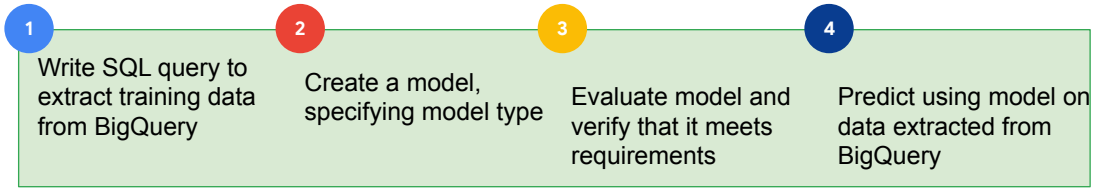
very common to enrich your data with pre-trained models, to take advantage of unstructured data

Build your own model

Developers
Data Analysts
ML engineers

Google Cloud

Working with BigQuery ML



Google Cloud

Where was this article published?

1. Techcrunch
2. GitHub
3. NY Times

Unlikely Partnership in House Gives Lawmakers Hope for Border Deal

Representatives Nita M. Lowey and Kay Granger are the first women to lead the House Appropriations Committee. Their bond gives lawmakers optimism for the work to come.

By EMILY COCHRANE

Fitbit's newest fitness tracker is just for employees and health insurance members

1 hour ago **Jon Russell**

Fitbit has a new fitness tracker, but it's one that you can't buy in stores. The company quietly uncorked the Inspire on Friday, releasing its first product that is available only to co...

Downloading the Android Studio Project Folder

FTC Engineering edited this page on Sep 19, 2017 · 1 revision

Downloading the Android Studio Project Folder

Google Cloud

SQL query to extract data

```
SELECT
  url, title
FROM
  `bigquery-public-data.hacker_news.stories`
WHERE
  LENGTH(title) > 10
  AND LENGTH(url) > 0
LIMIT 10
```

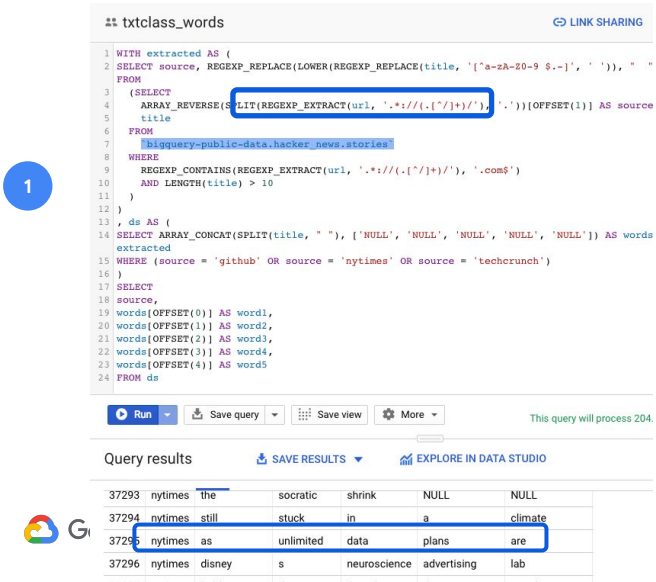
1

url	title
http://www.bbc.co.uk/news/business-27732743	Vodafone reveals direct government wiretaps
https://www.kickstarter.com/projects/appdocu/a...	Doc – App: The Human Story
http://www.starwebworld.com/android-jelly-bean...	Android Jelly Bean: Streaming Audio Through th...
http://www.myplanetdigital.com/digital_strateg...	Why Canadian Tech Entrepreneurs Need to Man/Wo...
http://startupislandconference.com/index.html	StartupConference June 13. - 16. 2013, HVAR Cr...
http://kopimism.org/	Kopimism Hactivism Meetup Tomorrow (Sunday) in...
http://unearthedgadget.com/xbox-live-gold-2/14...	Xbox Live Gold Membership Is It Really Worth -...
https://evertale.com	Evertale changes the way people remember
http://www.racketboy.com/retro/commodore-amiga...	Commodore Amiga: A Beginner's Guide
http://www.extremetech.com/extreme/156393-cold...	Cold fusion reactor "independently verified"

Google Cloud


```
SELECT
  url, title
FROM
  `bigquery-public-data.hacker_news.stories`
WHERE
  LENGTH(title) > 10
  AND LENGTH(url) > 0
LIMIT 10
```

Use regex to get source + train on words of title




```
WITH extracted AS (
SELECT source, REGEXP_REPLACE(LOWER(REGEXP_REPLACE(title,
'[^a-zA-Z0-9 $.-]', ' ')), "  ", " ") AS title FROM
  (SELECT
    ARRAY_REVERSE(SPLIT(REGEXP_EXTRACT(url, '.*://(.[^/]+)/'), '.'))[OFFSET(1)]
AS source,
    title
  FROM
    `bigquery-public-data.hacker_news.stories`
  WHERE
    REGEXP_CONTAINS(REGEXP_EXTRACT(url, '.*://(.[^/]+)/'), '.com$')
    AND LENGTH(title) > 10
  )
)
, ds AS (
SELECT ARRAY_CONCAT(SPLIT(title, " "), ['NULL', 'NULL', 'NULL', 'NULL', 'NULL'])
AS words, source FROM extracted
WHERE (source = 'github' OR source = 'nytimes' OR source = 'techcrunch')
)
SELECT
source,
words[OFFSET(0)] AS word1,
words[OFFSET(1)] AS word2,
words[OFFSET(2)] AS word3,
```

```
words[OFFSET(3)] AS word4,
words[OFFSET(4)] AS word5
FROM ds
```

Create classification model

2

```
CREATE OR REPLACE MODEL advdata.txtclass
OPTIONS(model_type='logistic_reg', input_label_cols=['source'])
AS

WITH extracted AS (
...
)
, ds AS (
SELECT ARRAY_CONCAT(SPLIT(title, " "), ['NULL', 'NULL', 'NULL',
'NULL', 'NULL']) AS words, source FROM extracted
WHERE (source = 'github' OR source = 'nytimes' OR source =
'techcrunch')
)
SELECT
source,
words[OFFSET(0)] AS word1,
words[OFFSET(1)] AS word2,
words[OFFSET(2)] AS word3,
words[OFFSET(3)] AS word4,
words[OFFSET(4)] AS word5
FROM ds
```

Query to extract training data

Google Cloud

A model feels like just another table that is being created.

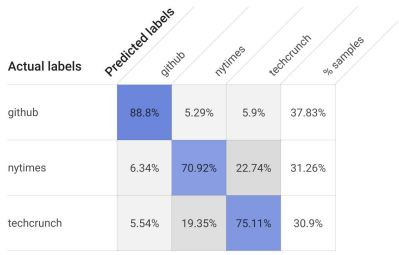
Evaluate model

```
SELECT * FROM ML.EVALUATE(MODEL advdata.txtclass)
```

3

precision	recall	accuracy	f1_score	log_loss	roc_auc
0.783	0.783	0.79	0.783	0.858	0.918

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



Google Cloud

SELECT * FROM ML.EVALUATE(MODEL advdata.txtclass)

Predict using trained model

4

```
SELECT * FROM ML.PREDICT(MODEL advdata.txtclass,(
     SELECT 'government' AS word1, 'shutdown' AS word2, 'leaves'
AS word3, 'workers' AS word4, 'reeling' AS word5
     UNION ALL SELECT 'unlikely', 'partnership', 'in', 'house',
'gives'
     UNION ALL SELECT 'fitbit', 's', 'fitness', 'tracker', 'is'
     UNION ALL SELECT 'downloading', 'the', 'android', 'studio',
'project'
))
```

Row	predicted_source	word1	word2	word3	word4	word5
1	nytimes	government	shutdown	leaves	workers	reeling
2	nytimes	unlikely	partnership	in	house	gives
3	techcrunch	fitbit	s	fitness	tracker	is
4	techcrunch	downloading	the	android	studio	project

"Batch prediction"

Google Cloud


```
SELECT * FROM ML.PREDICT(MODEL advdata.txtclass,(
     SELECT 'government' AS word1, 'shutdown' AS word2, 'leaves' AS word3,
'workers' AS word4, 'reeling' AS word5
     UNION ALL SELECT 'unlikely', 'partnership', 'in', 'house', 'gives'
     UNION ALL SELECT 'fitbit', 's', 'fitness', 'tracker', 'is'
     UNION ALL SELECT 'downloading', 'the', 'android', 'studio', 'project'
))
```

Linear Classifier (Logistic regression)

```
create or replace model models.will_buy_banana_example
options(model_type='logistic_reg', input_label_cols=['banana']) AS

with purchases_data AS (
select
  CAST(user_id AS string) customer_id,
  CAST(zip_code AS string) home_zipcode,
  ['dawn', 'morning', 'afternoon', 'night'][OFFSET(CAST
(TRUNC(order_hour_of_day/6) AS INT64))] AS time_of_day,
  (SELECT 24852 IN (SELECT product_id FROM UNNEST(order_lines))) AS
banana
FROM operations.orders_with_lines
JOIN operations.customers_loyalty
ON user_id = id
)
select * from purchases_data
```

Google Cloud

Show them running this model

DNN Classifier

```
create or replace model models.will_buy_banana_example
options(model_type='dnn_classifier', hidden_units=[64, 8],
input_label_cols=['banana']) AS

with purchases_data AS (
select
  CAST(user_id AS string) customer_id,
  CAST(zip_code AS string) home_zipcode,
  ['dawn', 'morning', 'afternoon', 'night'][OFFSET(CAST
(TRUNC(order_hour_of_day/6) AS INT64))] AS time_of_day,
  (SELECT 24852 IN (SELECT product_id FROM UNNEST(order_lines))) AS
banana
FROM operations.orders_with_lines
JOIN operations.customers_loyalty
ON user_id = id
)
select * from purchases_data
```

Google Cloud

Show them running this model

There are two types of DNN models available as alpha in Qwiklabs (this will work in Qwiklabs but not your own accounts yet as of Sept 2019). The dnn_classifier and the dnn_regressor as we will see shortly.

Linear regression

```
CREATE OR REPLACE MODEL ch09edu.bicycle_model
OPTIONS(input_label_cols=['duration'],
        model_type='linear_reg')
AS

SELECT
  duration
  , start_station_name
  , CAST(EXTRACT(dayofweek from start_date) AS STRING)
        as dayofweek
  , CAST(EXTRACT(hour from start_date) AS STRING)
        as hourofday
FROM
  `bigquery-public-data.london_bicycles.cycle_hire`
```

Google Cloud

Here is a simple forecasting model to predict the duration of a bike ride in london. We are using Linear Regression as our model type and our label is 'duration' as you see here.


```
CREATE OR REPLACE MODEL ch09edu.bicycle_model
OPTIONS(input_label_cols=['duration'],
        model_type='linear_reg')
AS

SELECT
  duration
  , start_station_name
  , CAST(EXTRACT(dayofweek from start_date) AS STRING)
        as dayofweek
  , CAST(EXTRACT(hour from start_date) AS STRING)
        as hourofday
FROM
  `bigquery-public-data.london_bicycles.cycle_hire`
```

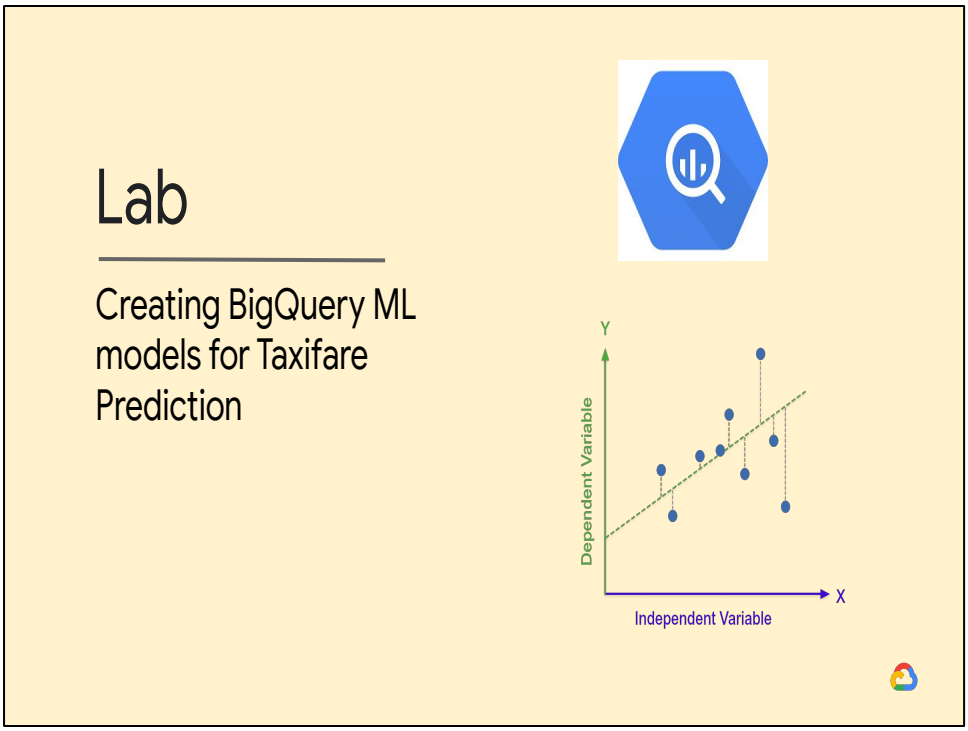
DNN regression

```
CREATE OR REPLACE MODEL ch09edu.bicycle_model
OPTIONS(input_label_cols=['duration'],
        model_type='dnn_regressor', hidden_units=[32, 4])
AS

SELECT
  duration
  , start_station_name
  , CAST(EXTRACT(dayofweek from start_date) AS STRING)
        as dayofweek
  , CAST(EXTRACT(hour from start_date) AS STRING)
        as hourofday
FROM
  `bigquery-public-data.london_bicycles.cycle_hire`
```

Google Cloud

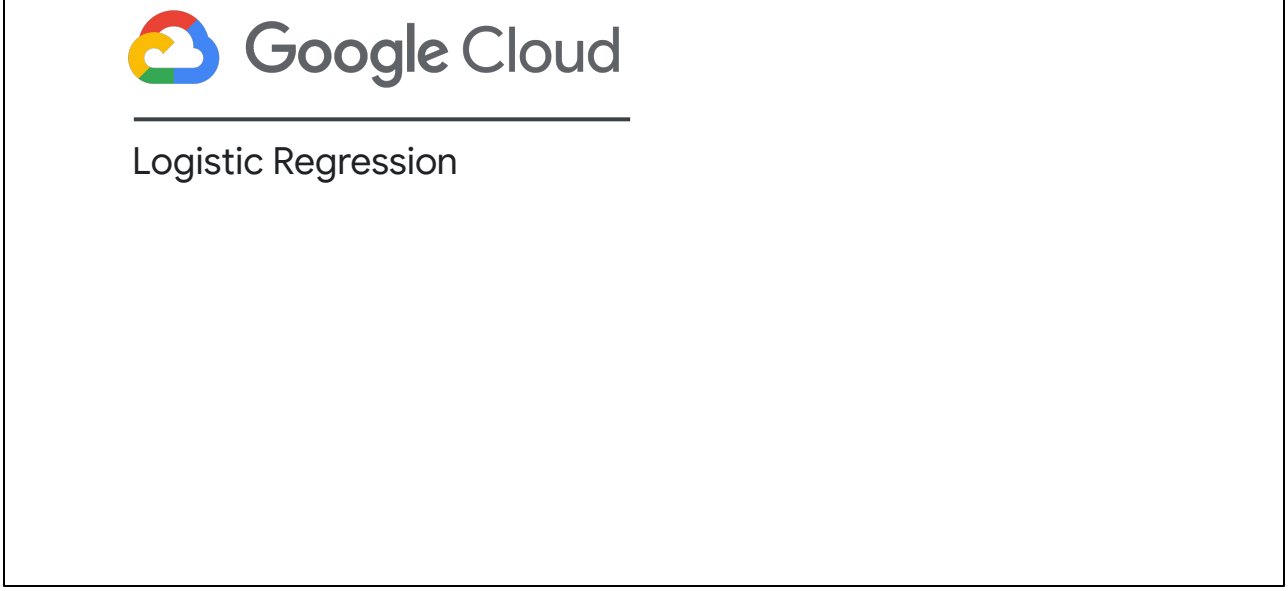
And here is the DNN_regressor that you will be creating as part of your lab. Note the model type and hidden units parameters in model options.

```
CREATE OR REPLACE MODEL ch09edu.bicycle_model
OPTIONS(input_label_cols=['duration'],
        model_type='dnn_regressor', hidden_units=[32, 4])
AS

SELECT
  duration
  , start_station_name
  , CAST(EXTRACT(dayofweek from start_date) AS STRING)
        as dayofweek
  , CAST(EXTRACT(hour from start_date) AS STRING)
        as hourofday
FROM
  `bigquery-public-data.london_bicycles.cycle_hire`
```



In this lab, you will build and train a linear regression model in BigQueryML

Link to lab: Creating BigQuery ML models for Taxifare Prediction (qwiklabs). github repo link here.



Now that we have learned about L1 regularization, let's dive deeper into logistic regression and see why it is important to use regularization.

Suppose you use linear regression to predict coin flips

You might use features like angle of bend, coin mass, etc. What could go wrong?

Suppose we want to predict the outcomes of coin flips. We all know that for a fair coin, the expected value is 50% heads and 50% tails. What if we had instead an unfair coin like with a bend in it. Let's say we want to generalize coin flip prediction to all coins: fair and unfair, big and small, heavy and light, etc. What features could we use to predict whether a flip would be heads or tails?

Perhaps we could use the angle of the bend because it distributes X% of mass in the other dimension and/or creates a difference in rotation due to air resistance or center of mass. The mass of the coin might also be a good feature to know, as well as size properties such as diameter, thickness, etc. We could do some feature engineering on this to get the volume of the coin and furthermore the density. Maybe the type of material or materials the coin is composed of would be useful information.

These features would be pretty easy to measure however they are only one side of the coin, pun intended, of the coin flip, the rest comes down to the action of the flip itself such as how much linear and angular velocity the coin was given, the angle of launch, the angle of what it lands on, wind speed, etc. These might be a bit harder to measure.

Suppose you use linear regression to predict coin flips

What could go wrong?

Now that we have all of these features, what's the simplest model we could use to predict heads or tails? Linear regression of course. What could go wrong with this choice though?

Our labels are heads or tails, or thought of in another way heads or not heads which we can represent with a one-hot encoding of 1 for heads and 0 for not-heads. But if we use linear regression with the standard mean squared error loss function, our predictions could end up being outside the range of (0, 1). What does it mean if we predict 2.75 for the coin flip state? That makes no sense.

A model that minimizes squared error is under no constraint to treat this as a probability in (0, 1), but this is what we need here. In particular, you could imagine a model that predicts values less than 0 or greater than 1 for some new examples. This would mean we can't use this model as a probability.

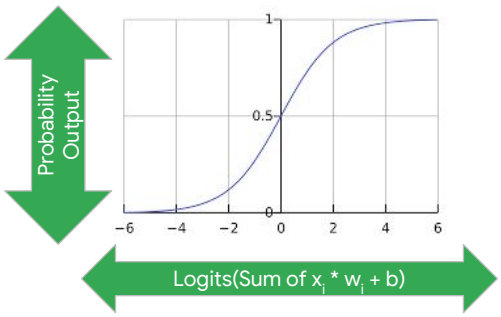
Simple tricks like capping the predictions at 0 or 1 would introduce bias. So, we need something else -- in particular, a new loss function.

Logistic Regression: transform linear regression by a sigmoid activation function

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x}+b)}}$$

linear model

squish through a sigmoid



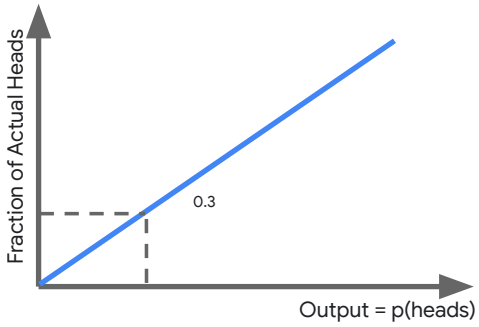
Converting this from linear regression to logistic regression can solve this dilemma. From an earlier course of ours, we went through the history of ML and introduced the Sigmoid activation function. Let's take a deeper look into that now.

The sigmoid activation function essentially takes the weighted sum, w transpose x plus b, from a linear regression and instead of just outputting that and then calculating the mean squared error loss, we change the activation function from linear to sigmoid which takes that as an argument and squashes it smoothly between 0 and 1. The input into the sigmoid, normally the output of a linear regression, is called the logit. So we are performing a non-linear transformation of our linear model.

Notice how the probability asymptotes to zero when the logits go to negative infinity and to one when the logits go to positive infinity. What does this imply for training? Unlike mean squared error, the sigmoid never guesses 1.0 or 0.0 probability. This means, that in gradient descent's constant drive to get the loss closer and closer to zero, it will drive the weights closer and closer to plus or minus infinity, in the absence of regularization, which can lead to problems.

\hat{y}=\frac{1}{1+e^{-(\mathbf{w}^{T}\mathbf{x}+b)}}

The output of Logistic Regression is a calibrated probability estimate



Useful because we can cast binary classification problems into probabilistic problems:

Will customer buy item?

becomes

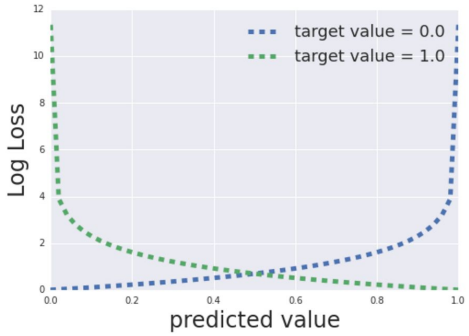
Predict the probability that customer buys item

First though, how can we interpret the output of a sigmoid? Is it just some function that's range is 0 to 1, of which there are many, or is it something more? The good news is it is something more, it is a calibrated probability estimate. Beyond just the range, the sigmoid function is the cumulative distribution function of the logistic probability distribution whose quantile function is the inverse of the logit which models the log odds. Therefore, mathematically the outputs of a sigmoid can be considered probabilities.

In this way, we can think of calibration as the fact that the outputs are real world values like probabilities. This is in contrast to uncalibrated outputs like an embedding vector, which is internally informative, but the values have no real world correlation. Lots of output activation functions, in fact an infinite number, could give you a number between 0 and 1 but only the sigmoid is proven to be a calibrated estimate of the training dataset probability of occurrence.

Using this fact about the sigmoid activation function, we can cast binary classification problems into probabilistic problems. For instance, instead of a model just predicting a yes or no, such as will a customer buy an item, it can now predict the probability that a customer buys an item. This paired with a threshold can provide a lot more predictive power than just a simple binary answer.

Typically, use cross-entropy (related to Shannon's information theory) as the error metric



Less emphasis on errors where the output is relatively close to the label.

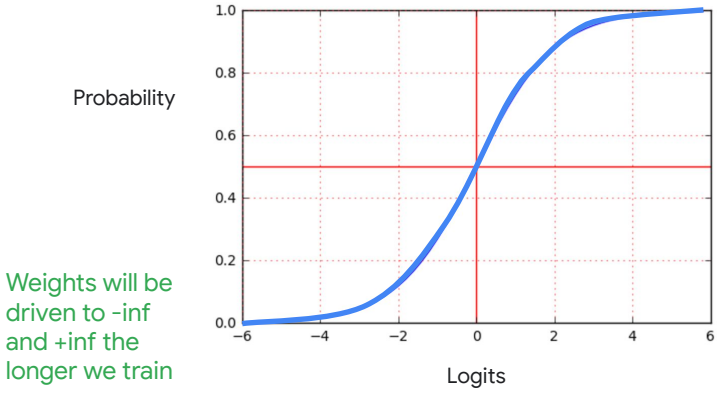
$$LogLoss = \sum_{(\boldsymbol{x},y)\in D} -ylog(\hat{y}) - (1-y)log(1-\hat{y})$$

However, regularization is important in logistic regression because driving the loss to zero is difficult and dangerous.

First, as gradient descent seeks to minimize cross entropy it pushes output values closer to 1 for positive labels and closer to 0 for negative labels. Due to the equation of the sigmoid, the function asymptotes to 0 when the logit is negative infinity and to 1 when the logit is positive infinity. To get the logits to negative or positive infinity, the magnitude of the weights is increased and increased leading to numerical stability problems: overflows and underflows. This is dangerous and can ruin our training.

https://pixabay.com/en/architecture-building-infrastructure-2559480/ (cc0)

Regularization is important in logistic regression because driving the loss to zero is difficult and dangerous



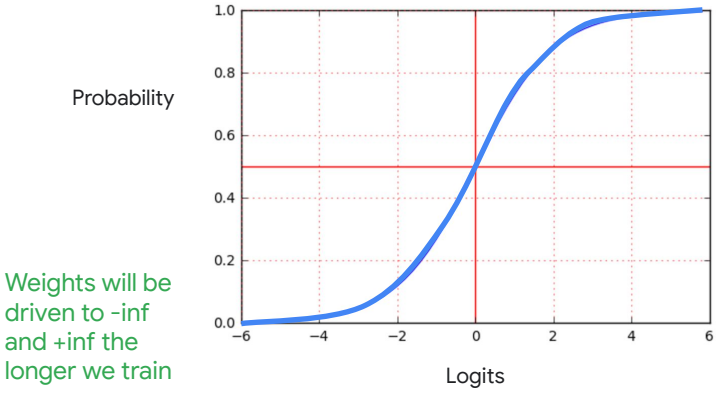
Weights will be driven to -inf and +inf the longer we train

Also, near the asymptotes, as you can see from the graph, the sigmoid function becomes flatter and flatter. This means that the derivative is getting closer and closer to zero. Since we use the derivative in backpropagation to update the weights, it is important for the gradient not to become zero, or else training will stop. This is called saturation when all activations end up in these plateaus which leads to a vanishing gradient problem and makes training difficult.

There is also a potentially useful insight here. Imagine you assign a unique id for each example, and map each id to its own feature. If you use un-regularized logistic regression, this will lead to absolute overfitting, as the model tries to drive loss to zero on all examples and never gets there, the weights for each indicator feature will be driven to +inf or -inf. This can happen in practice in high dimensional data with feature crosses. Often there's a huge mass of rare crosses that happens only on one example each. So how can we protect ourselves from overfitting?

https://pixabay.com/en/architecture-building-infrastructure-2559480/ (cc0)

Regularization is important in logistic regression because driving the loss to zero is difficult and dangerous



Also, near the asymptotes, as you can see from the graph, the sigmoid function becomes flatter and flatter. This means that the derivative is getting closer and closer to zero. Since we use the derivative in backpropagation to update the weights, it is important for the gradient not to become zero, or else training will stop. This is called saturation when all activations end up in these plateaus which leads to a vanishing gradient problem and makes training difficult.

There is also a potentially useful insight here. Imagine you assign a unique id for each example, and map each id to its own feature. If you use un-regularized logistic regression, this will lead to absolute overfitting, as the model tries to drive loss to zero on all examples and never gets there, the weights for each indicator feature will be driven to +inf or -inf. This can happen in practice in high dimensional data with feature crosses. Often there's a huge mass of rare crosses that happens only on one example each. So how can we protect ourselves from overfitting?

https://pixabay.com/en/architecture-building-infrastructure-2559480/ (cc0)

Logistic Regression Regularization Quiz

Why is it important to add regularization to logistic regression?

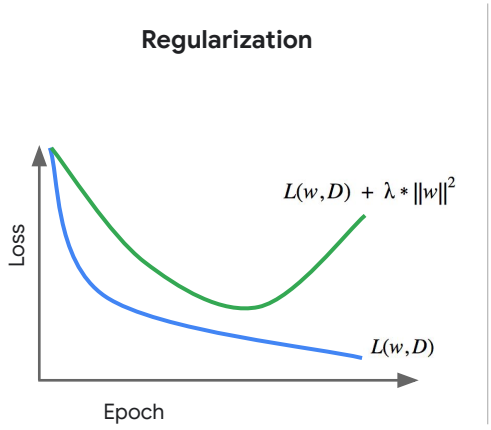
A. Helps stops weights being driven to +/- infinity.
B. Helps logits stay away from asymptotes which can halt training
C. Transforms outputs into a calibrated probability estimate
D. Both A & B
E. Both A & C

Ryan:
Question: Which of these is important when performing logistic regression?

Answer: The correct answer is both A & B. Adding regularization to logistic regression helps keep the model simpler by having smaller parameter weights. This penalty term added to the loss function makes sure that cross entropy through gradient descent doesn't keep pushing the weights from going closer and closer to plus or minus infinity and casuing numerical issues. Also, with now smaller logits, we can now stay in the less flat portions of the sigmoid function making our gradients less close to zero and thus allowing weight updates and training to continue.

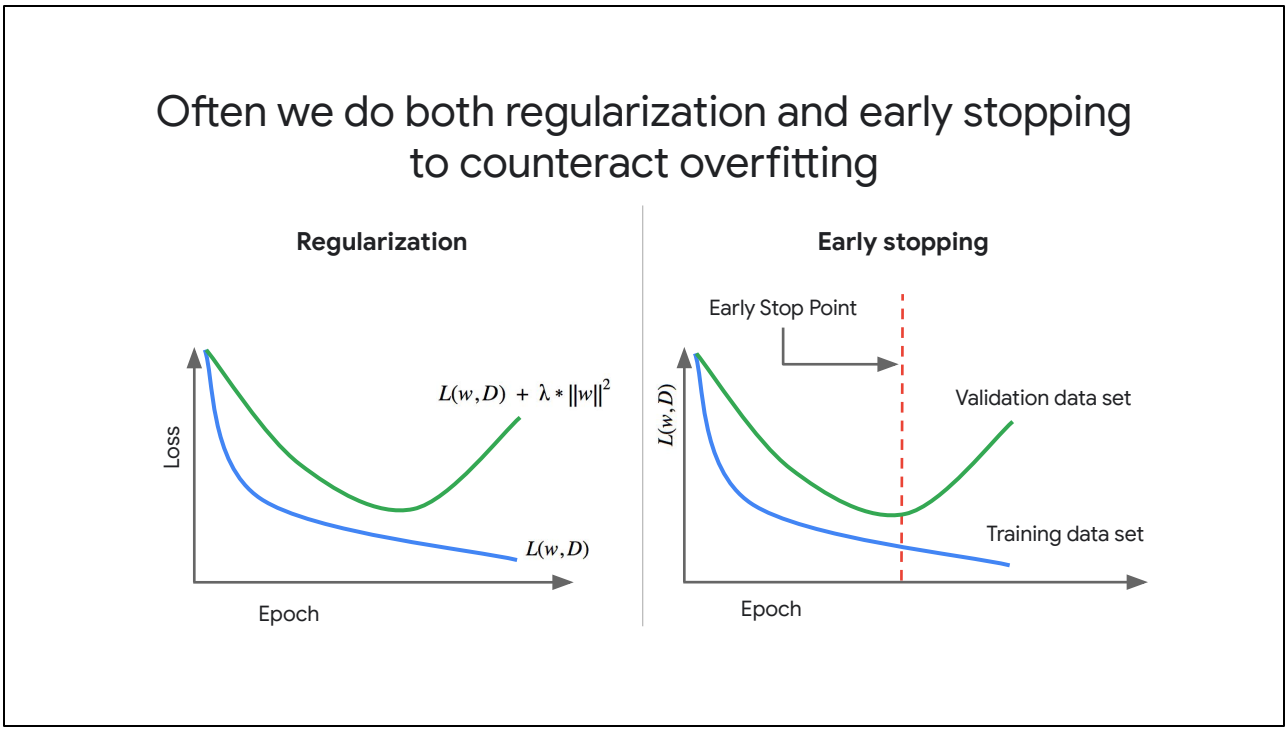
C is incorrect and therefore so is E because regularization does not transform the outputs into a calibrated probability estimate. The great thing about logistic regression is that it already outputs a calibrated probability estimate since the sigmoid function is the cumulative distribution function of the logistic probability distribution. This allows us to actually predict probabilities instead of just binary answers like yes or no, true or false, buy or sell, etc.

Often we do both regularization and early stopping to counteract overfitting



To counteract overfitting we often do both regularization and early stopping.

For regularization, model complexity increases with large weights, and so as we tune and start to get larger and larger weights for rarer and rarer scenarios, we end up increasing the loss, so we stop. L2 regularization will keep the weight values smaller and L1 regularization will make the model sparser by dropping poor features. To find the optimal L1 and L2 hyperparameter choices during hyperparameter tuning, you are searching for the point in the validation loss function where you obtain the lowest value. At that point any less regularization increases your variance, starts overfitting, and hurts your generalization and any more regularization increases your bias, starts underfitting, and hurts your generalization.



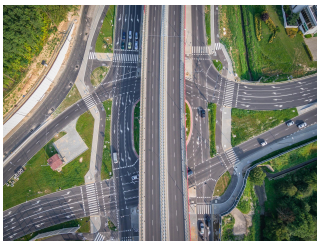
Early stopping stops training when overfitting begins. As you train your model, you should evaluate your model on your validation dataset every so many steps, epochs, minutes, etc. As training continues both the training error and the validation error should be decreasing but at some point the validation error might begin to actually increase! It is at this point that the model is beginning to memorize the training dataset and lose its ability to generalize to the validation dataset, and most importantly, to the new data that we eventually want to use this model for. Using early stopping would stop the model at this point and then backup and use the weights from the previous step before it hit the validation error inflection point. Here, the loss is just L(w,D) i.e. no regularization term. Interestingly, early stopping is an approximate equivalent of L2 regularization and is often used in its place because it is computationally cheaper.

Fortunately, in practice, we always use both explicit regularization (L_1 and L_2), and also some amount of early stopping regularization. Even though L2 regularization and early stopping seem a bit redundant, for real world systems, you may not quite choose the optimal hyperparameters and thus early stopping can help fix that choice for you.

In many real-world problems, the probability is not enough; we need to make a binary decision

Send the mail to spam folder or not?

Approve the loan or not?



Which road should we route the user through?

It's great that we can obtain a probability from our logistic regression model, however at the end of the day, sometimes users just want a simple decision to be made for them for their real world problems. Should the email be sent to the spam folder or not? Should the loan be approved or not? Which road should we route the user through?

How can we use our probability estimate to help the tool, using our model, to make a decision? We choose a threshold!

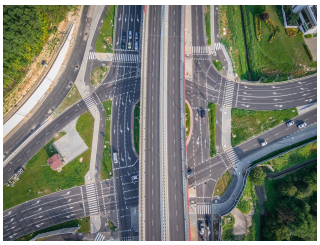
https://pixabay.com/en/the-intersection-way-investment-2683894/ (cc0)
https://pixabay.com/en/thumb-hand-arm-guide-guiding-grip-422558/ (cc0)

https://pixabay.com/en/spam-mail-email-mailbox-garbage-964521/(cc0)

In many real-world problems, the probability is not enough; we need to make a binary decision

Send the mail to spam folder or not?

Approve the loan or not?



Which road should we route the user through?

Choice of threshold is important and can be tuned

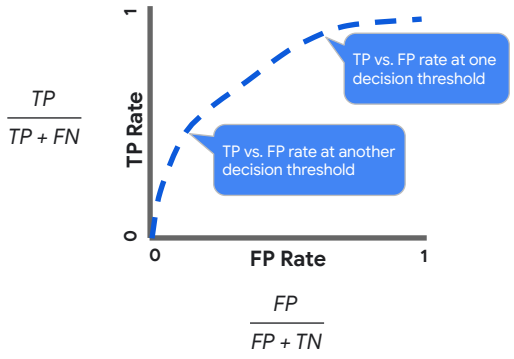
A simple threshold of a binary classification problem would be all probabilities less than or equal to 50% should be a no and all probabilities greater than 50% should be a yes. However, for certain real world problems we may want a different split like 60/40, 20/80, 99/1 etc. depending on how we want our balance of our type I and type II errors or in other words our balance of false positives and false negatives.

For binary classification, we will have four possible outcomes: true positives, true negatives, false positives, and false negatives. Combinations of these values can lead to evaluation metrics like precision which is the number of true positives divided by all positives and recall which is the number of true positives divided by the sum of true positives and false negatives which gives the sensitivity or true positive rate. You can tune your choice of threshold to optimize the metric of your choice. Is there an easy way to help us do this?

https://pixabay.com/en/the-intersection-way-investment-2683894/ (cc0)
https://pixabay.com/en/thumb-hand-arm-guide-guiding-grip-422558/ (cc0)

https://pixabay.com/en/spam-mail-email-mailbox-garbage-964521/(cc0)

Use the ROC curve to choose the decision threshold based on decision criteria



A Receiver Operating Characteristic curve, or ROC curve for short, shows how a given model's predictions create different true positive vs. false positive rates when different decision thresholds are used.

As we lower the threshold, we are likely to have more false positives, but will also increase the number of true positives we find. Ideally, a perfect model would have zero false positives and zero false negatives which plugging that into the equations would give a true positive rate of 1 and a false positive rate of 0.

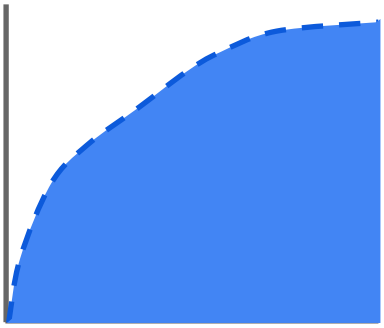
To create a curve, we would pick each possible decision threshold and re-evaluate. Each threshold value creates a single point, but by evaluating many thresholds eventually a curve is formed. Fortunately, there's an efficient sorting-based algorithm to do this.

Each model will create a different ROC curve, so how can we use these curves to compare the relative performance of our models when we don't know exactly what decision threshold we want to use?

The Area-Under-Curve (AUC) provides an aggregate measure of performance across all possible classification thresholds

AUC helps you choose between models when you don't know what decision threshold is going to be ultimately used.

"If we pick a random positive and a random negative, what's the probability my model scores them in the correct relative order?"



We can use the area under the curve as an aggregate measure of performance across all possible classification thresholds. AUC helps you choose between models when you don't know what decision threshold is going to be ultimately used. It is like asking "if we pick a random positive and a random negative, what's the probability my model scores them in the correct relative order?"

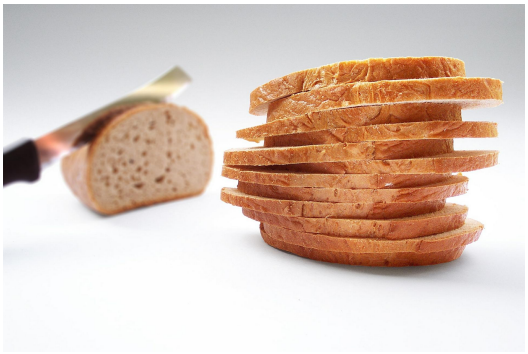
The nice thing about AUC is that it's scale invariant and classification threshold invariant. People like to use it for those reasons. People sometimes also use AUC for the precision-recall curve, or more recently precision-recall-gain curves which just use different combinations of the four prediction outcomes as metrics along the axes.

However, treating this only as an aggregate measure can mask some effects. For example, a small improvement in AUC might come by doing a better job of ranking "very unlikely negatives" as "even still yet more unlikely", which is fine but potentially not materially beneficial.

Logistic Regression predictions should be unbiased

average of predictions == average of observations

Look for bias in slices of data, this can guide improvements.

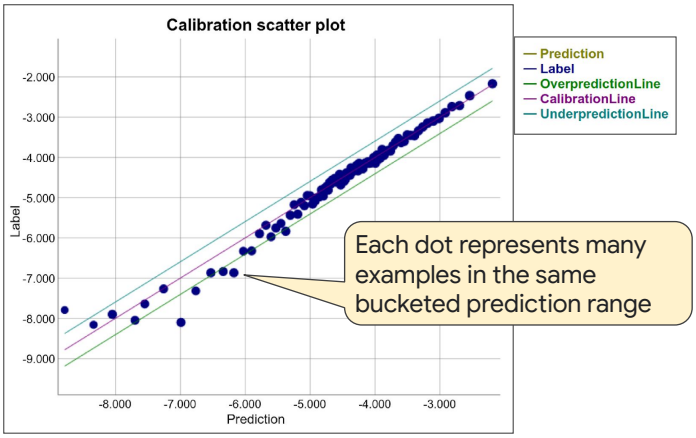


When evaluating our logistic regression models, we need to make sure predictions should be unbiased. When we talk about bias in this sense we are not talking about the bias term in the model's linear equation. Instead we mean, there should not be an overall shift in either the positive or negative direction. A simple way to check the prediction bias is to compare the average value of predictions made by the model over a dataset to the average value of the labels in that dataset. If they are not relatively close then you might have a problem.

Bias is like a canary in a mine where we can use it as an indicator of something being wrong. If you have bias, you definitely have a problem, but even zero bias alone does not mean everything in your system is perfect, but it is a great sanity check. If you have bias, you could have an incomplete feature set, a buggy pipeline, a biased training sample, etc. You can look for bias in slices of data which can help guide improvements of removing bias from your model. Let's look at an example of how you can do that.

https://pixabay.com/en/bread-slice-of-bread-knife-cut-534574/ (cc0)

Use calibration plots of bucketed bias to find slices where your model performs poorly



Here's a calibration plot from the sibyl experiment browser. You'll notice that this is on a log / log scale, as we're comparing the bucketized log-odds predicted to the bucketized log-odds observed.

You'll note that things are pretty well calibrated in the moderate range, but the extreme low end is pretty bad.

This can happen when parts of the data space is not well represented, or because of noise, or because of overly strong regularization.

The bucketing can be done in a couple of ways. You can bucket by linearly breaking up the target predictions, or you can bucket by quantiles.

Why do we need to bucket prediction to make calibration plots when predicting probabilities?

For any given event, the true label is either 0 or 1, for example, not clicked or clicked. But our prediction values will always be a probabilistic guess somewhere in the middle, like 0.1 or 0.33. For any individual example, we're always off. But if we group enough examples together, we'd like to see that on average the sum of the true 0's and 1's is about the same as the mean probability we're predicting.

Logistic Regression Quiz

Which of these is important when performing logistic regression?

A. Adding regularization
B. Choosing a tuned threshold
C. Checking for bias
D. All of the above

Ryan:
Question: Which of these is important when performing logistic regression?

Answer: The correct answer is all of the above. It is extremely important that our model generalizes so that we have the best predictions on new data which is the entire reason we created the model to begin with. To help do this, it is important that we do not overfit our data, therefore adding penalty terms to the objective function like with L1 regularization for sparsity and L2 regularization for keeping model weights small and adding early stopping can help in this regard.

It is also important to choose a tuned threshold for deciding what decisions to make with your probability estimate outputs to minimize or maximize the business metric that is important to you. If this isn't well defined, then we can use more statistical means such as calculating the number of true and false positives and negatives and combining them into different metrics such as the true and false positive rates. We can then repeat this process for many different thresholds and then plot the area under the curve, or AUC, to come up with a relative aggregate measure of model performance.

Lastly, it is important that our predictions are unbiased and even if there isn't bias, we should still be dilligent to make sure our model is performing well. We can begin looking for bias by making sure that the average of the predictions is very close to the average of observations. A helpful way to find where bias might be hiding is to look at slices of data and use something like a calibration plot to isolate the problem areas for

further refinement.

So far, we've talked about ML strategy -- about what machine learning means, what problems it can solve, and how to put it into practice at your company. Besides these technical and business aspects, another aspect that you want to keep in mind is that whatever ML models you build have to treat all your customers fairly. A key aspect of your ML strategy has to be ... building machine learning systems in an inclusive way.

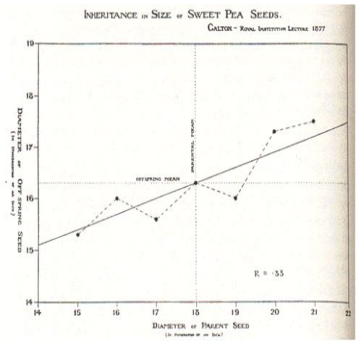
In this module, I will show you how to identify the origins of bias in machine learning -- it comes down to the training data. Then, I will show you ways in which you can apply an inclusive lens throughout the machine learning development process—from the data exploration, all the way over to evaluating the performance of your trained model.
We will also discuss practical techniques to make your models inclusive.

Let's delve in.

Linear regression was invented when computations were done by hand, but it continues to work well for large datasets

Linear Regression
For predicting planets and pea growth

1800s



Linear regression was “invented” for predicting the movement of planets and the size of pea pods based on their parents.
Sir Francis Galton pioneered the use of statistical methods to measurements of natural phenomena.
He was looking at data on the relative sizes of parents and their children in various species, including sweet peas.
He observed something that is not immediately obvious ... something rather strange.
Sure, a larger-than-average parent tends to produce a larger-than-average child.
But how much larger is that child to the average of other children in its generation?
It turned out that this ratio for the child is *less* than the corresponding ratio for the parent.

For example, if the parent's size is 1.5 standard deviations from the mean within its own generation, then you would predict that the child's size will be less than 1.5 standard deviations from the mean within its cohort.

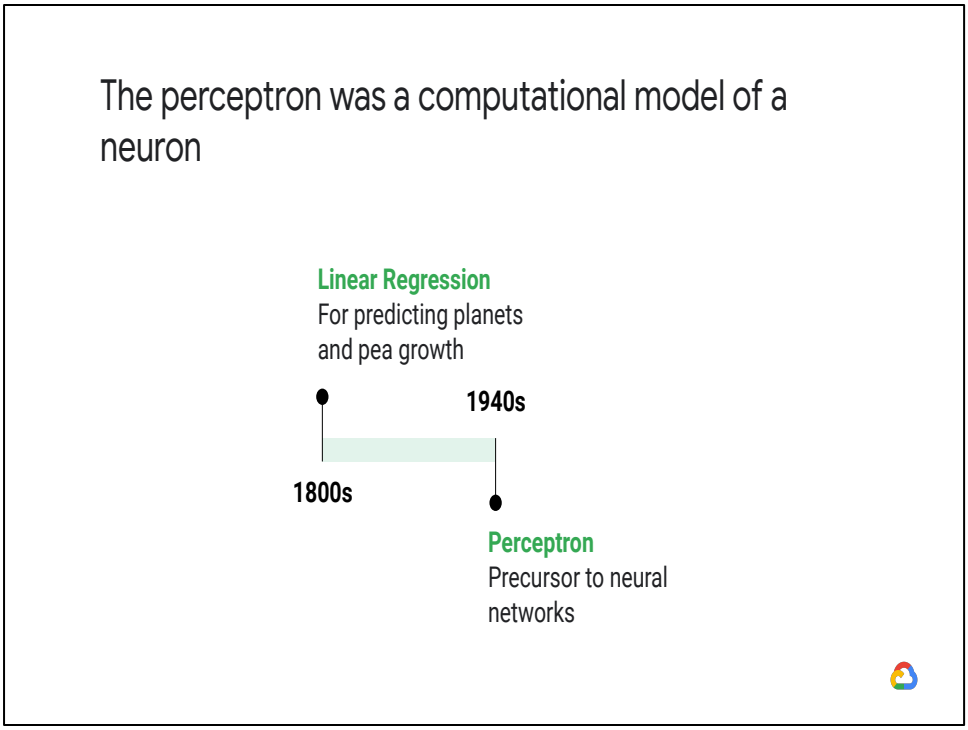
We say that generation-by-generation, things in nature regress, or go back, to the mean. Hence the name “linear regression”. This chart here, from 1877, is the first ever linear regression. Pretty cool!

Compute power in the 1800s was somewhat limited, so they didn’t even realize how well this would continue to work once we had large datasets.

There is actually a closed form solution for solving linear regression, but gradient

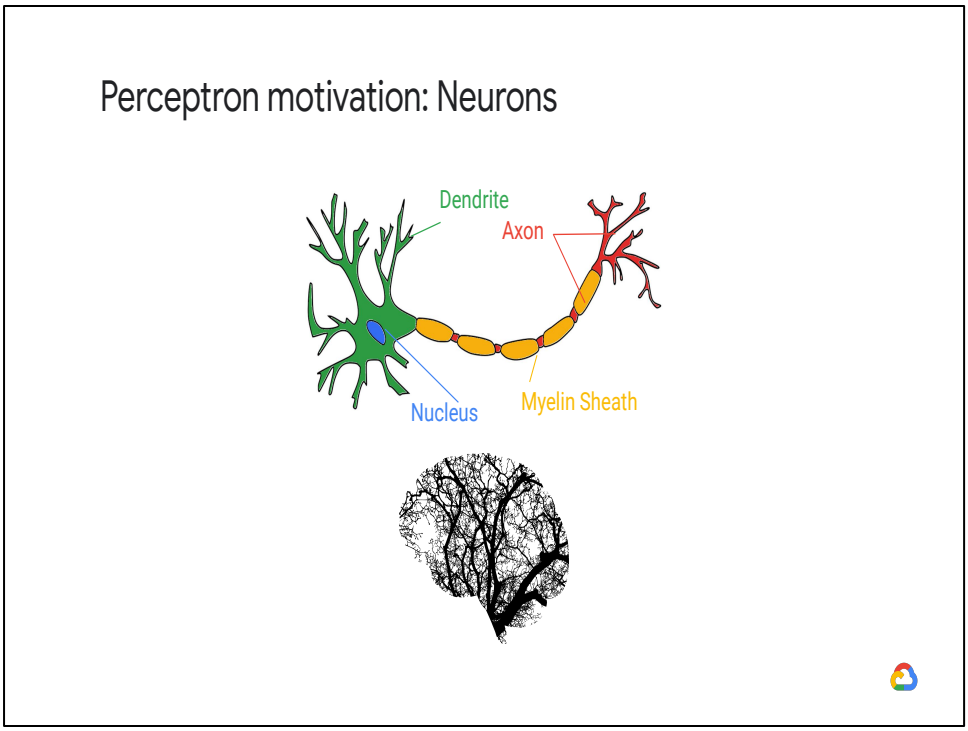
descent methods can also be used, each with their pros and cons, depending on your dataset. Let's look under the hood on how linear regression works.

Image from http://people.duke.edu/~rnau/regintro.htm, but is of book that is out of copyright.



So, linear regression was pretty much it as far as learning from data was concerned until the 1940s ...

A researcher, Frank Rosenblatt, comes up with the perceptron as a computational model of a neuron in the human brain and shows how it can “learn” simple functions. It is what we would call today a binary linear classifier, where we are trying to find a single line that splits the data into two classes. A single layer of perceptrons would be the simplest possible feedforward neural network. Inputs would feed into a single layer of perceptrons, and a weighted sum would be performed. This sum would then pass through what we call today an activation function, which is just a mathematical function you apply to each element that is now residing within that neuron. Remember though at this point, this is still just a linear classifier, so the activation function, which is linear, in this case just returns its inputs. Comparing the output of this to a threshold would then determine which class each point belongs to. The errors would be aggregated and used to change the weights used in the sum, and the process would happen again until convergence.



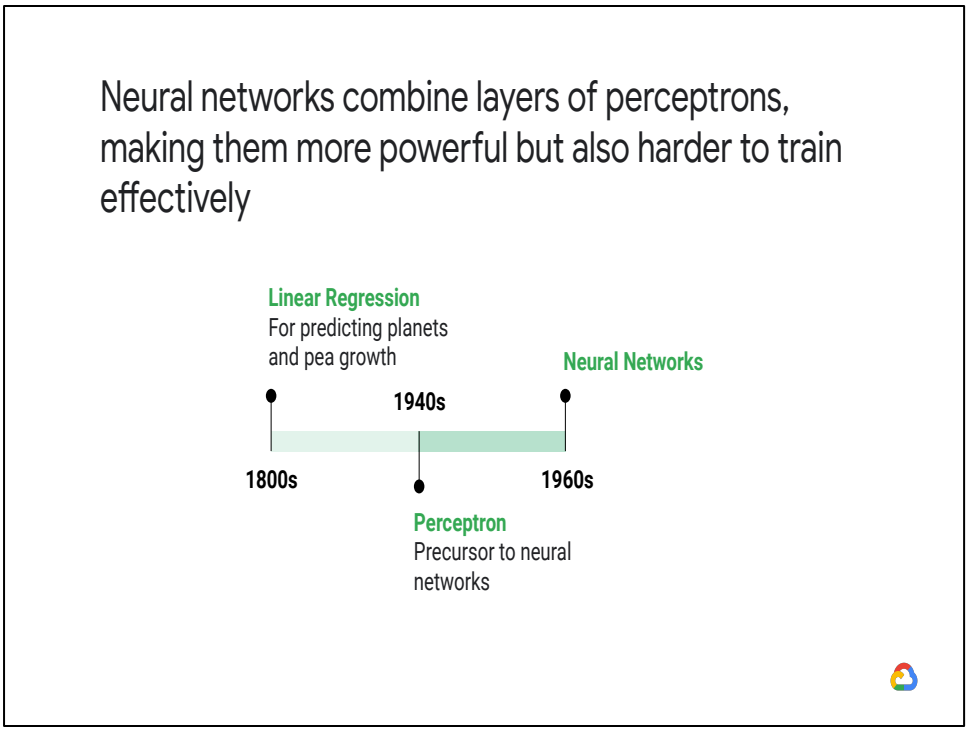
If you are trying to come up with a simple model of something that learns a desired output from a given input distribution then you needn't look far since our brains do this all day long making sense out of the world around us and all of the signals our bodies receive. One of the fundamental units of the brain is the neuron. Neural networks are just groups of neurons connected together in different patterns or architectures. A biological neuron has several components specialized in passing along electrical signal which allows you and I to have thoughts, perform actions, and study the fascinating world of machine learning.

Electrical signals from other neurons, such as sensory neurons in the retina of your eye, are propagated from neuron to neuron. The input signal is received at one end of the neuron which is made up of dendrites. These dendrites might not just collect electrical signal from just one other neuron but possibly from several which all get summed together over windows in time that changes the electrical potential of the cell. A typical neuron has a resting electric potential of -70mV. As the input stimuli received at the dendrites increases, eventually it reaches a threshold around -55mV in which case a rapid depolarization of the axon occurs with a bunch of voltage gates opening up allowing a sudden flow of ions. This causes the neuron to fire an action potential of electric current along the axon, aided by the myelin sheath for better transmission to the axon terminals. Here neurotransmitters are released at synapses that then travel across the synaptic cleft to usually the dendrites of other neurons. Some of the neurotransmitters are excitatory where they raise the potential of the next cell while some are inhibitory and lower the potential. The neuron repolarizes to an even lower potential than resting for a refractory period and then the process

continues at the next neuron until maybe it eventually reaches a motor neuron and moves your hand to shield the sun out of your eyes.

So what does all of this biology and neuroscience have to do with machine learning?

Graphic: Ryan, Modified to make a Google Themed Neuron:
https://pixabay.com/en/neuron-nerve-cell-axon-dendrite-296581/ cc0
Graphic: (human brain) cc0:
https://pixabay.com/en/brain-anatomy-abstract-art-2146817/

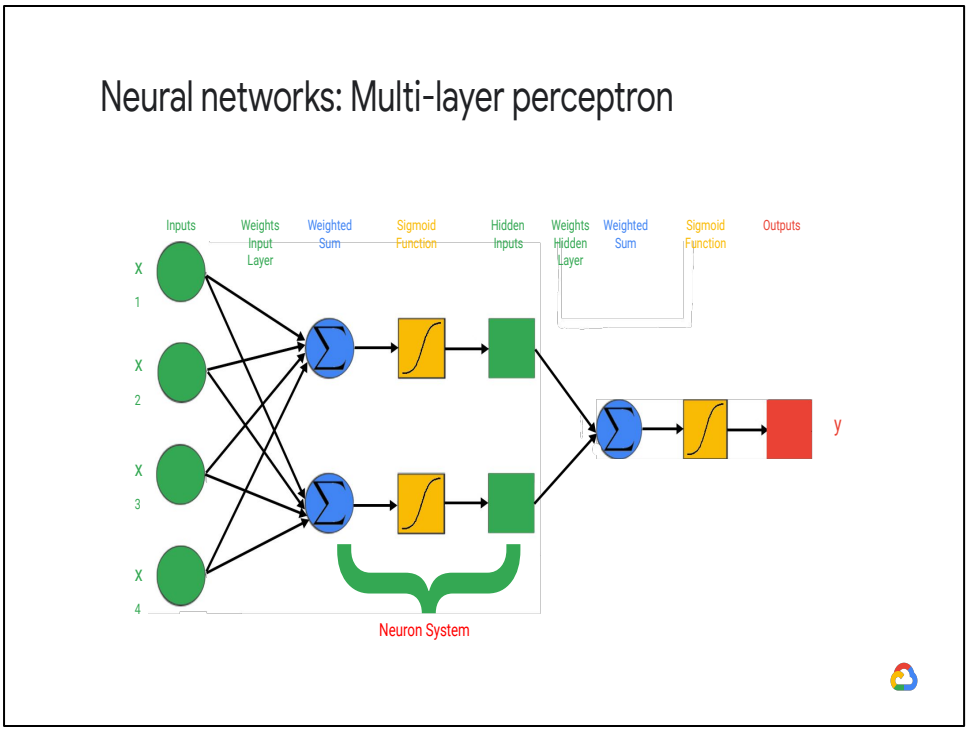


Why only one layer of perceptrons? Why not send the output of one layer as the input to the next layer?
Combining multiple layers of perceptrons sounds like it would be a much more powerful model.

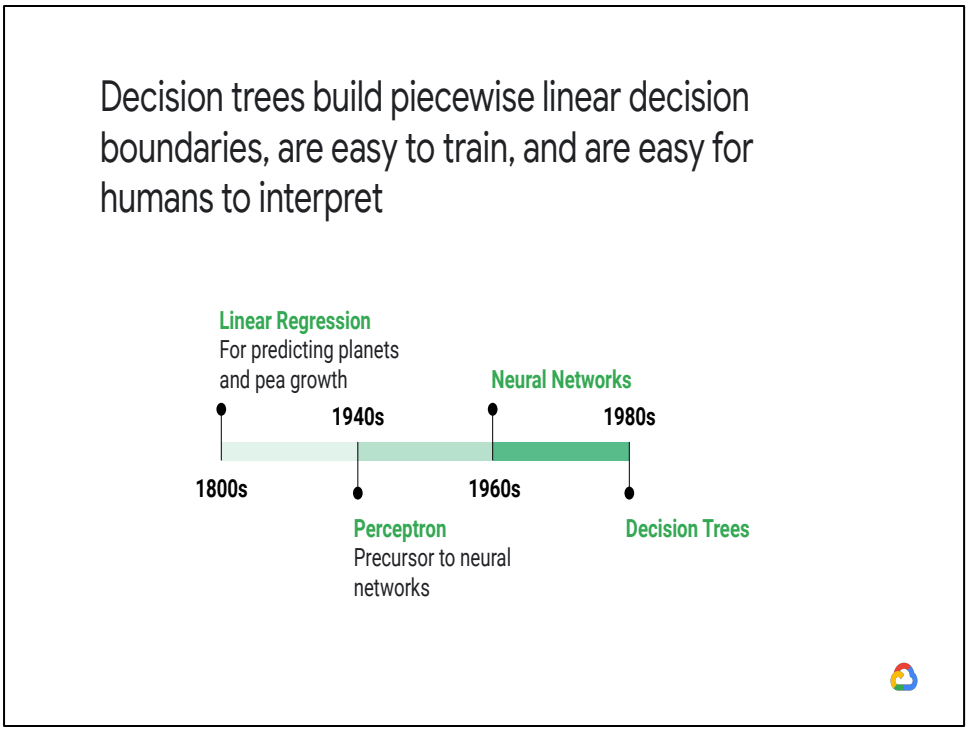
However without using nonlinear activation functions, all of the additional layers can be compressed back down into just a single linear layer, and there is no real benefit. You need nonlinear activation functions.
Therefore, sigmoid or tanh activation functions started to be used for nonlinearity. At the time, we were limited to just these because we needed a differentiable function since that fact is exploited in backpropagation to update the model weights.

Modern-day activation functions are not differentiable, and people didn't know how to work with them.
This constraint -- that activation functions had to be differentiable -- could make the networks hard to train.
The effectiveness of these models was also constrained by the amount of data, available computational resources, and other difficulties in training.
For instance, optimization tended to get caught in saddle points instead of finding the global minimum we hoped it would during gradient descent.

However, once the trick to use Rectified Linear Units, or ReLUs, was developed, then you could have faster training like 8 to 10X; almost guaranteed convergence for logistic regression.

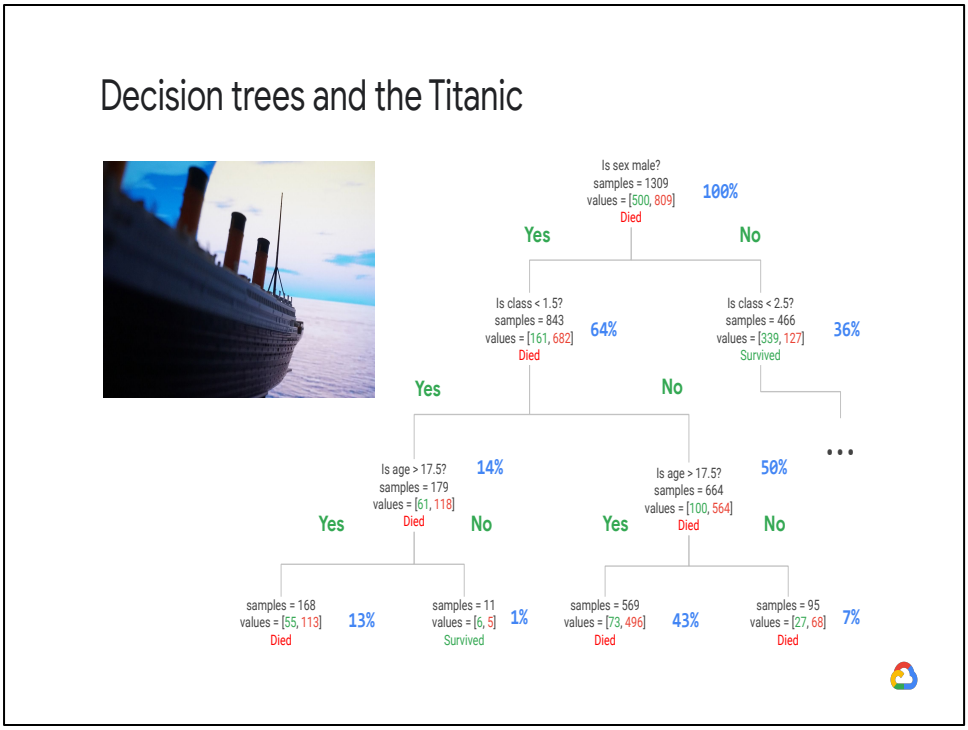


Building off of the perceptron, just like the brain, we can connect many of them together to form layers to create feedforward neural networks. Really not much has changed in components from the single layer perceptron. There are still inputs, weighted sums, activation functions, and outputs. One difference is that the inputs to neurons not in the input layer are not the raw inputs but the outputs of the previous layer. Another difference is that the weights connecting the neurons between layers are no longer a vector but now a matrix because of the completely connected nature of all neurons between layers. For instance, in the diagram the input layer weights matrix is 4 x 2 and the hidden layer weights matrix is 2 x 1. We will learn later that neural networks don't always have complete connectivity which has some amazing applications and performance like with images. Also, there are different activation functions than just the unit step function such as the sigmoid and the hyperbolic tangent or tanh activation functions. Each non-input neuron, you can think of as a collection of three steps packaged up into a single unit. The first component is a weighted sum, the second component is the activation function, and the third component is the output of the activation function.



Decision tree algorithms such as ID3 and C4.5 are invented in the 80s/90s. They are better at certain types of problems than linear regression and are very easy for humans to interpret. Finding the optimal splitting when creating the trees is an NP-hard problem, therefore greedy algorithms were used to hopefully construct trees as close to optimal as possible.

They create a piecewise linear decision surface which is essentially what a layer of ReLUs gives you. But with DNNs or Deep Neural Networks, each of the ReLU layers combines to make a hyper planar decision surface, which can be much more powerful. But I am skipping ahead to why DNNs can be better than decision trees. Let's first talk about decision trees



Decision trees are one of the most intuitive machine learning algorithms. They can be used for both classification and regression. Imagine you have a dataset and you want to determine how the data is all split into different buckets. The first thing you should do is brainstorm some interesting questions to query the dataset with. Let's walk through an example. Here is the well known problem of predicting who lived or died in the Titanic catastrophe. There were people aboard from all walks of life, different backgrounds, different situations, etc. so we want to see if any of those possible features can partition my data in such a way that I can with high accuracy predict who lived.

A first guess at a feature could possibly be the sex of the passenger. Therefore I could ask the question, is the sex male? Thus I split the data with males going into one bucket and the rest going into another bucket. 64% of the data went into the male bucket leaving 36% going into the other one. Let's continue along the male bucket partition for now.

Another question I could ask is about what passenger class each passenger was. With our partitioning, now 14% of all passengers were male and of the lowest class whereas 50% of all passengers were male and of the higher two classes. The same type of partitioning could also continue in the female branch of the tree. Taking a step back, it is one thing for the decision tree building algorithm to split sex into two branches because there are only two possible values, but how did it decide to split passenger class with one passenger class branching to the left and two passenger classes branching to the right?

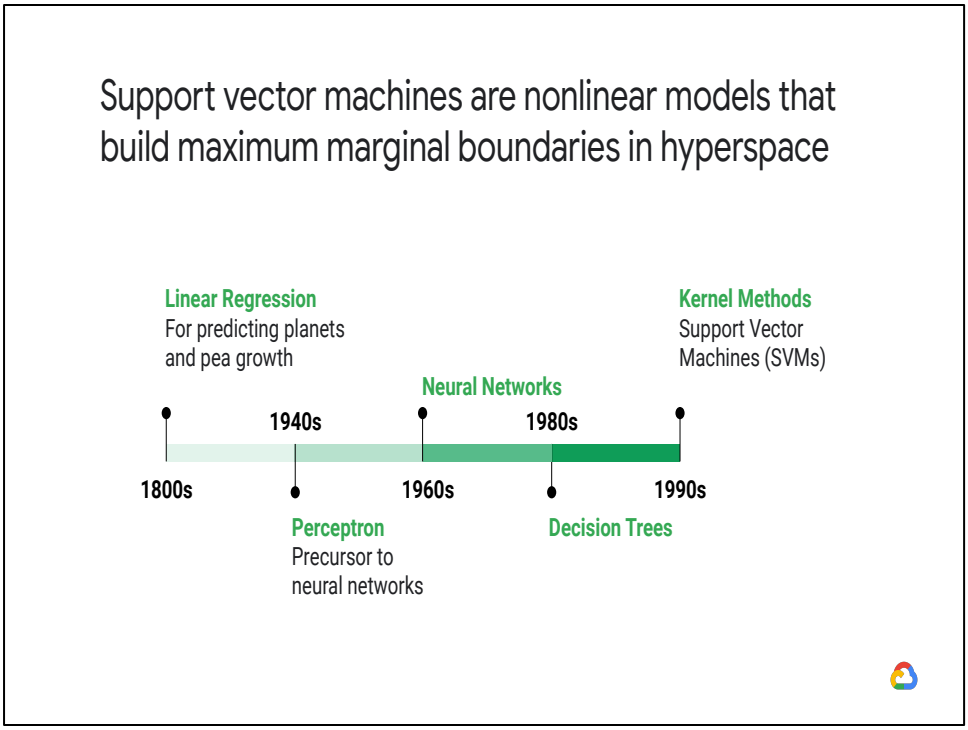
For instance, in the simple Classification And Regression Tree or CART algorithm, the algorithm tries to choose a feature and threshold pair that will produce the purest subsets when split. For classification trees, a common metric to use is the Gini impurity, but there is also entropy. Once it has found a good split, it searches for another feature threshold pair and splits that into subsets as well. This process continues on recursively until either the set maximum depth of the tree has been reached or if there are no more splits that reduce impurity. For regression trees, mean squared error is a common metric to split on.

Does this sound familiar how it chooses to split the data into two subsets? Each split is essentially just a binary linear classifier that finds a hyperplane that slices along one feature's dimension at some value which is the chosen threshold to minimize the members of a class falling on the other class' side of the hyperplane. Recursively creating these hyperplanes in a tree is analogous to layers of linear classifer nodes in a neural network. Very interesting! Now that we know how decision trees are built, let's continue building this tree a bit more.

Perhaps there is an age threshold that will help me split my data well for this classification problem. I could ask is the age greater than 17.5 years old? Looking at the lowest class branch of the male parent branch, now just 13% of the passengers were 18 and older while only 1% were younger. Looking at the classes associated with each node, only this one on the male branch so far is classified as survived. We could extend our depth and/or choose different features to hopefully keep expanding the tree until every node only has passengers that had survived or died.

However, there are problems with this because essentially I am just memorizing my data and fitting the tree perfectly to it. In practice, we are going to want to generalize this to new data and a model that has memorized the training set is probably not going to perform very well outside of it. There are some methods to regularize it, such as setting a minimum number of samples per leaf node, a maximum of leaf nodes, or a maximum number of features. You can also build the full tree and then prune unnecessary nodes.

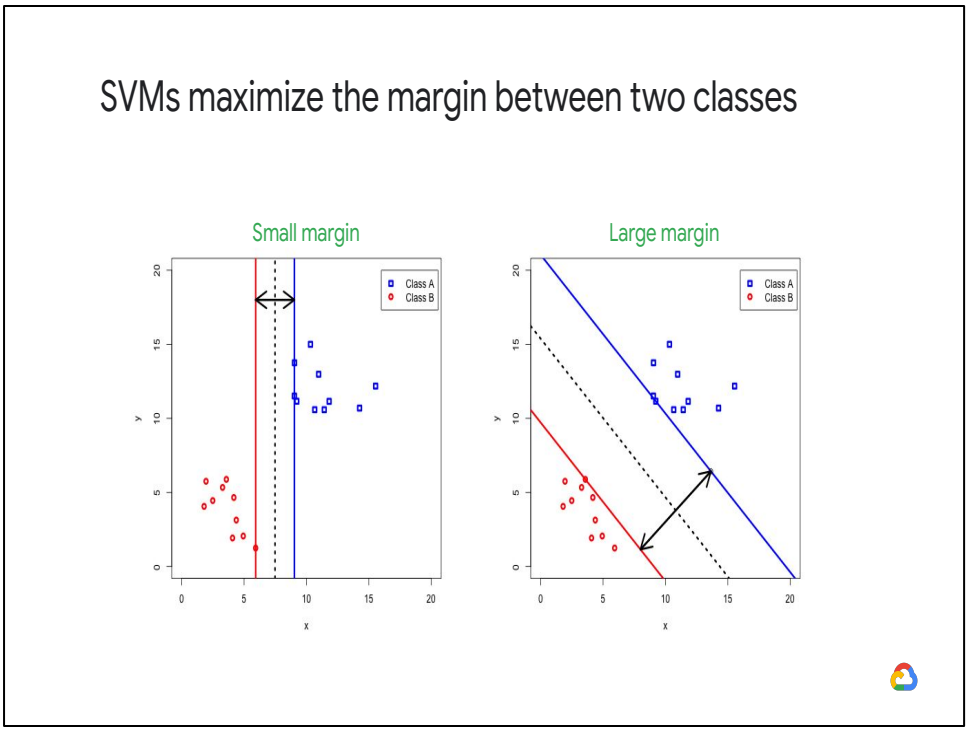
To really get the most out of trees it is usually best to combine them into forests which we will talk about very soon.



Starting in the 1990s, the field of kernel methods was formed. Corinna Cortes, Director of Google Research, was one of the pioneers.

This field of study allows interesting classes of new non-linear models, most prominently nonlinear SVMs or Support Vector Machines, which are maximum margin classifiers that you may have heard of before.

Fundamentally core to an SVM is a nonlinear activation plus a sigmoid output for maximum margins.

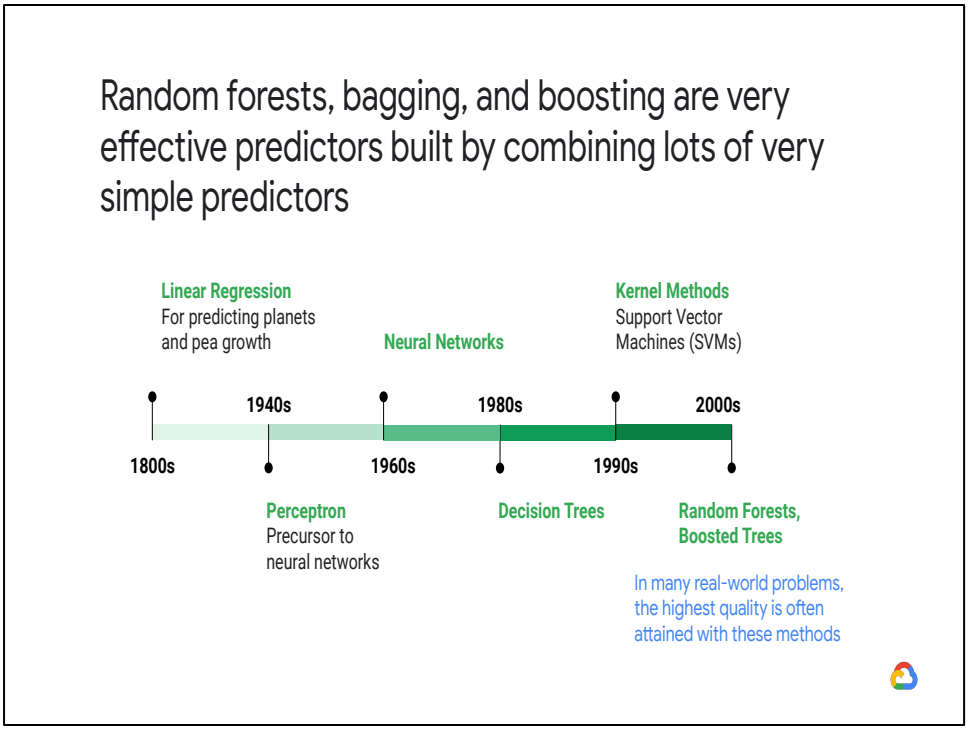


Earlier we have seen how logistic regression is used to create a decision boundary to maximize the log likelihood of the classification probabilities. In the case of a linear decision boundary, logistic regression wants to have each point of the associated class as far from the hyperplane as possible and provides a probability which can be interpreted as prediction confidence. There are an infinite number of hyperplanes you can create between two linearly separable classes such as the two hyperplanes shown as the dotted lines in the two figures here. In SVMs, we include two parallel hyperplanes on either side of the decision boundary hyperplane where they intersect with the closest data point on each side of the hyperplane. These are the support vectors. The distance between the two support vectors is the margin.

On the left we have a vertical hyperplane that indeed separates the two classes, however the margin between the two support vectors is small. By choosing a different hyperplane such as the one on the right, there is a much larger margin. The wider the margin, the more generalizable the decision boundary is which should lead to better performance on new data. Therefore, SVM classifiers aim to maximize the margin between the two support vectors using a hinge loss function compared to logistic regression's minimization of cross entropy.

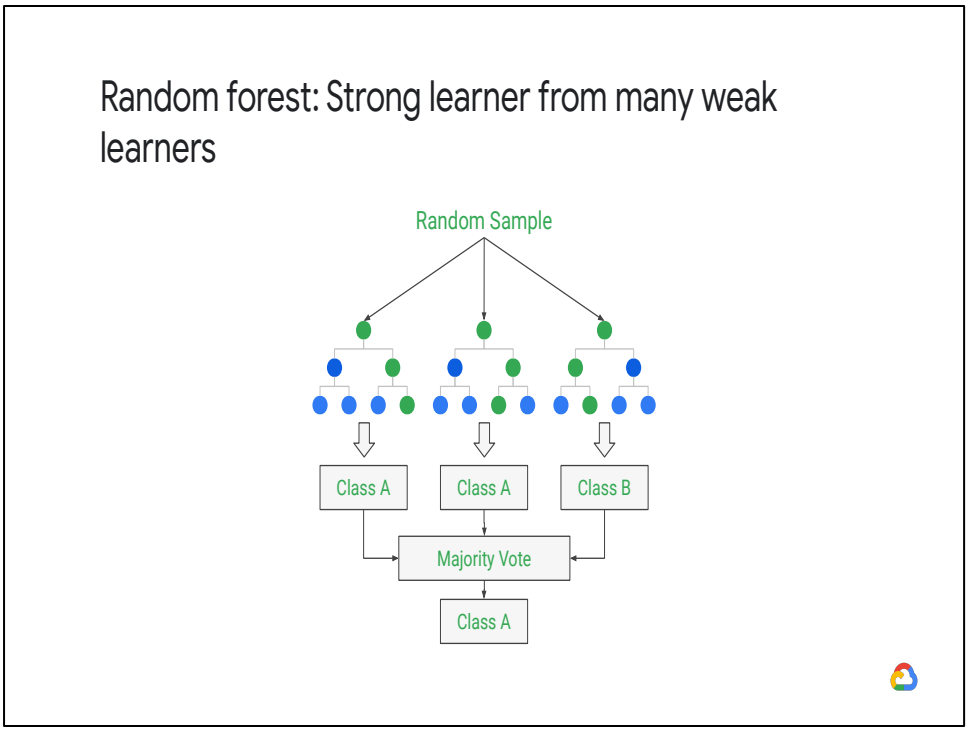
You might notice that I have only two classes which makes this is a binary classification problem. One class' label is given a value of 1 and the other class' label is given a value of -1. If there are more than two classes, then a one versus all approach should be taken and then choose the best out of the permuted binary classifications.

But what happens if the data is not linearly separable into the two classes?



Coming into the last few decades in the 2000s, machine learning research now had the computational power to combine and blend the performance across many models in what we call an ensemble method. You can imagine that, if the errors are independent for a number of simple weak learners, combined they would form a strong learner. DNNs can approximate this by using dropout layers, which help regularize the model and prevent overfitting. This can be simulated by randomly turning off neurons in the network with some probability for each forward pass, which will essentially be creating a “new” network each time.

In many of today’s real-world problems, the highest quality is often attained with these methods.



Oftentimes complex questions are better answered when aggregated from thousands of people's responses instead of those just by a sole individual. This is known as the wisdom of the crowd. The same applies to machine learning when you aggregate the results of many predictors, either classifiers or regressors. The group will usually perform better than the best individual model. This group of predictors is an ensemble which when combined in this way leads to ensemble learning. The algorithm that performs this learning is an ensemble method.

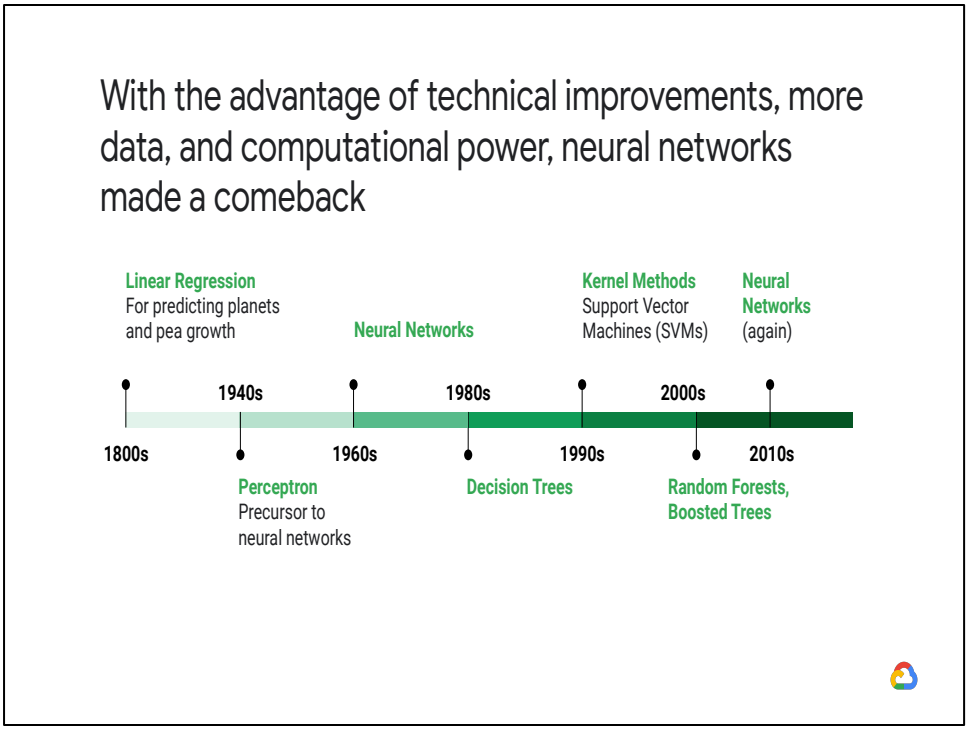
One of the most popular types of ensemble learning is the random forest. Instead of taking your entire training set and using that to build one decision tree, you could have a group of decision trees that each get a random subsample of the training data. Since they haven't seen the entire training set they can't have memorized the whole thing. Once all of the trees are trained on their subset of the data, you can now make the most important and valuable part of machine learning. Predictions! To do so, you would pass your test sample through each tree in the forest and then aggregate the results. If this is classification, there could be a majority vote across all trees which then would be the final output class. If it is regression, it could be an aggregate of the values such as the mean, max, median, etc.

To improve generalization, you can random sample the examples, and/or, the features. We call random sampling the examples with replacement, bagging, short for bootstrap aggregating, and pasting when without replacement. Each individual predictor has higher bias being trained on the smaller subset rather than the full dataset but the aggregation reduces both the bias and variance. This usually gives

the ensemble a similar bias as a single predictor on the entire training set, but with a lower variance. A great method of validation for your generalization error is to use your out-of-bag data, instead of having to have a separate set pulled from the dataset before training. It is reminiscent of k-fold validation using random holdouts. Random subspaces are made when we sample from the features, and, if we random sample examples too, it is called random patches.

Adaptive Boosting or AdaBoost and Gradient Boosting are both examples of boosting which is when we aggregate a number of weak learners to create a strong learner. Typically this is done by training each learner sequentially which tries to correct any issues the learner had before it. For boosted trees, as more trees are added to the ensemble the predictions usually improve. So do we continue to add trees ad infinitum? Of course not! You can use your validation set to use early stopping, so that we don't start overfitting our training data because we have added too many trees.

Lastly, just as we saw with neural networks, we can perform stacking where we can have meta learners learn what to do with predictions of the ensemble, which, can in turn also be stacked into meta meta learners and so on. We will see this subcomponent stacking and reuse in deep neural networks shortly.



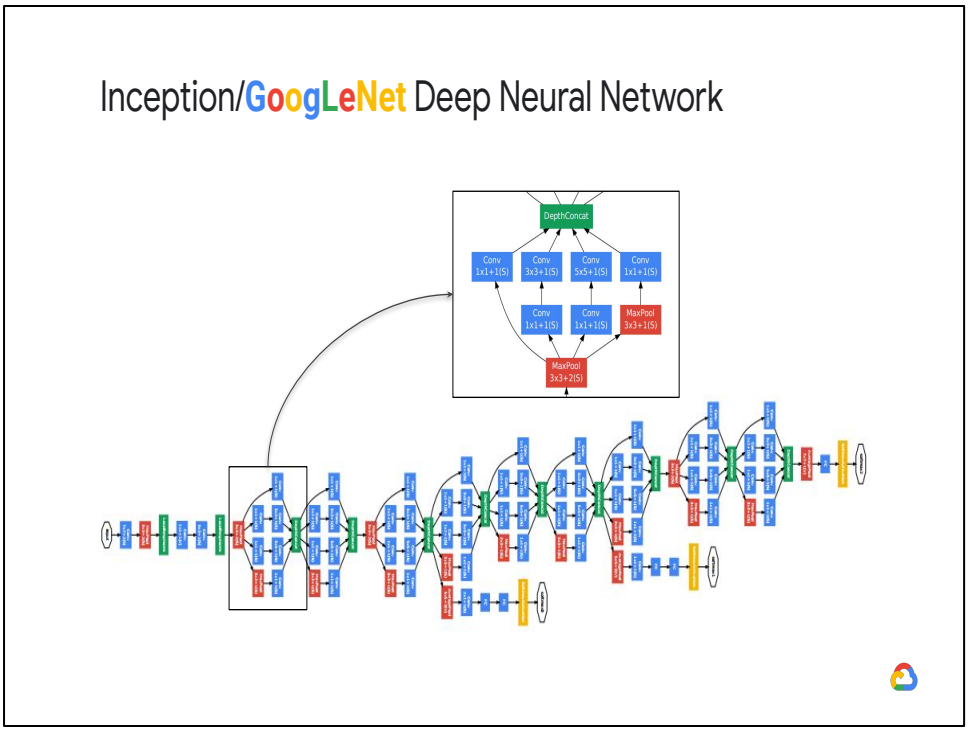
Back again on the timeline are neural networks, now with even more of an advantage through leaps in computational power and lots and lots of data. DNNs began to substantially outperform other methods on tasks such as Computer Vision.

In addition to the boon from boosted hardware, there were many new “tricks” and architectures that helped to improve trainability of deep neural networks like ReLUs, better initialization methods, CNNs or convolutional neural networks, and dropout.

We talked about some of these tricks from other ML models.

The use of non-linear activation functions like ReLUs which usually are set as the default now we talked about during the first look at neural networks.
Dropout layers began being used to help with generalization, which works like ensemble methods which we explored when talking about random forests and boosted trees.
Convolutional layers were added that reduce the computational and memory load due to their non-complete connectedness as well as being able to focus on local aspects for instance images rather than comparing unrelated things in an image.

In other words, all the advances that came about in other ML methods got folded back into neural networks. Let’s look at an example of a deep neural network.

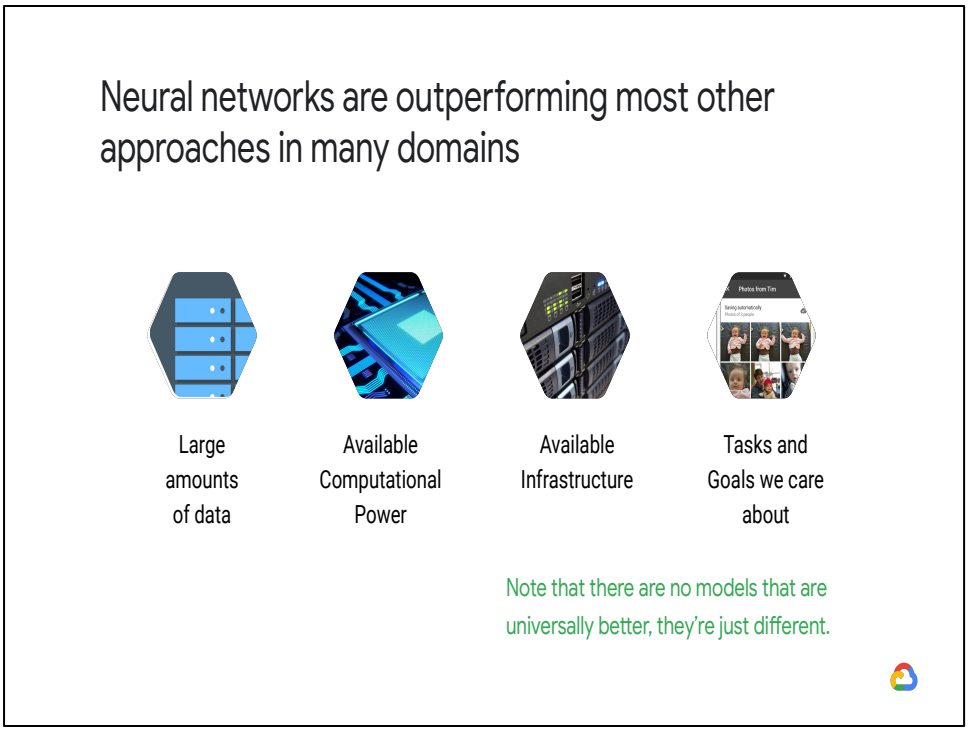


This exciting history of machine learning has culminated into deep learning with neural networks containing 100s of layers and millions of parameters, but with amazing results. Shown here is GoogLeNet or Inception which is an image classification model. It was trained for the ImageNet Large Visual Recognition Challenge in 2014 using data from 2012 where it has to classify images across 1000 classes with 1.2 million images for training. It has 22 deep layers, 27 if you include pooling which we will discuss in a later course, and 100 layers if you break it down into its independent building blocks. There are over 11 million trained parameters.

There are completely connected layers and some that aren't such as convolutional layers which we will talk about later. It used dropout layers to help generalize more, simulating an ensemble of deep neural networks. Just like we saw with neural networks and stacking, each box is a unit of components which is part of a group of boxes such as the one I have zoomed in on. This idea of building blocks adding up to something greater than the sum of its parts is one of the things that has made deep learning so successful. Of course an ever growing abundance of data and faster compute power and more memory helps too. There are now several versions beyond this that are much bigger and have even greater accuracy. The main takeaway from all of this history is that machine learning research reuses bits and pieces of techniques from other algorithms from the past to combine together to make ever powerful models and most importantly, experiment!

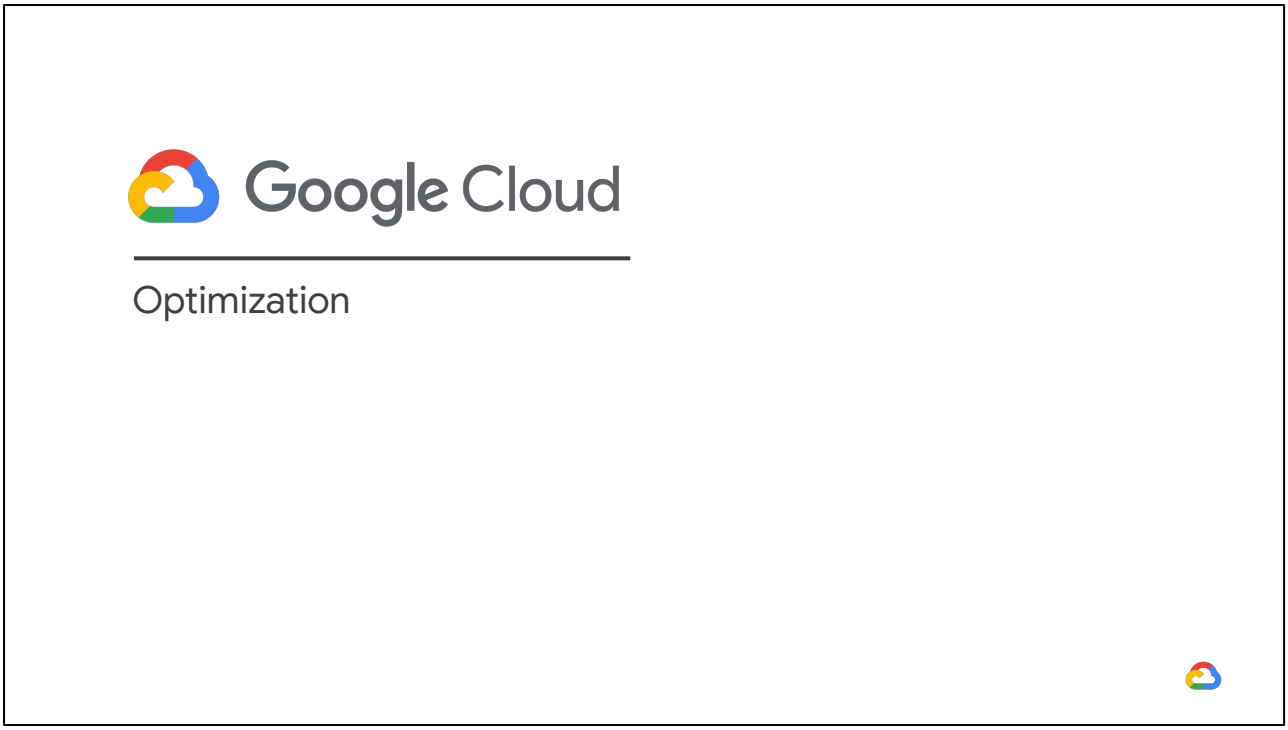
Inception: an image recognition model published by Google (From: Going deeper with convolutions, Christian Szegedy et al.)

Graphic: https://cloud.google.com/blog/big-data/2016/07/images/146798944178238/neural-networks-5.png



To recap, the last few decades have seen a proliferation in the adoption and performance of neural networks. With the ubiquity of data, these models have the benefit of more and more training examples to learn from. The increase in data and examples has been coupled with scalable infrastructure for even complex and distributed models with thousands of layers.

One note that we'll leave you with is that although performance with neural networks may be great for some applications, they are just one of the many types of models available for you to experiment with. Experimentation is key to getting the best performance using your data to solve your challenge.



Welcome to Optimization.

Machine Learning with TensorFlow on GCP

How Google does Machine Learning

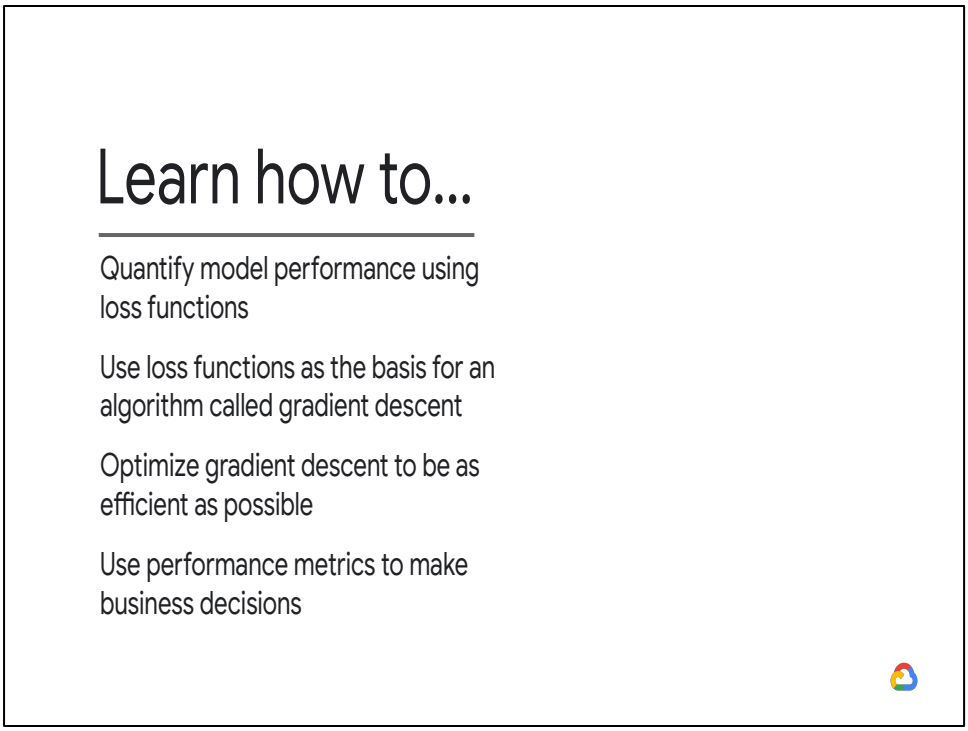
Launching into ML

Introduction to TensorFlow

Feature Engineering

The Art and Science of ML

This is the second module in the second chapter of the *Machine Learning with TensorFlow on GCP* specialization.

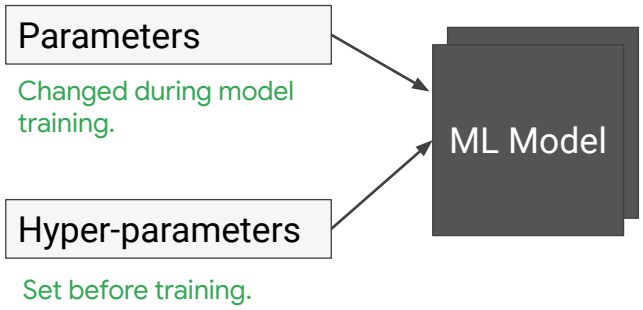


In this module, you'll learn how to measure model performance objectively using loss functions; use loss functions as the basis for an algorithm called gradient descent; optimize gradient descent to be as efficient as possible; and use performance metrics to make business decisions.

There are five main topics in this module.
First, you'll create a working but formal definition of what a model is.
Then, because optimization always requires a standard by which to say you're improving, loss functions will be discussed.
Then, you'll see how gradient descent is like trying to find the bottom of a hill defined by the loss function.
Next, you'll play around in a sandbox where you can see models descending loss surfaces in real time!
Lastly, you'll discuss how to measure a model's performance outside of the context of training.

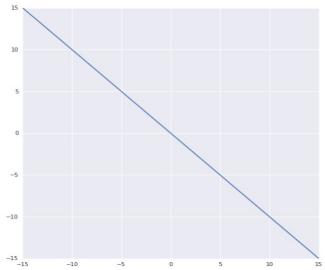
In this topic, you'll review exactly what an ML model is, and where parameters fit into the equation.

ML models are mathematical functions with parameters and hyper-parameters



ML models are mathematical functions with parameters and hyper-parameters. A parameter is a real-valued variable that changes during model training. A hyper-parameter is a setting that is set *before* training and which does not change afterwards.

Linear models have two types of parameters: Bias and weight



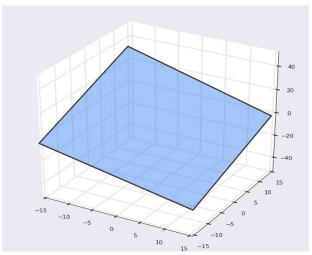
Linear

Output | Bias Term | Input | Weight

$$y = b + x \times m$$

$$y = b + X \times w$$

Model Parameters



Hyperplane

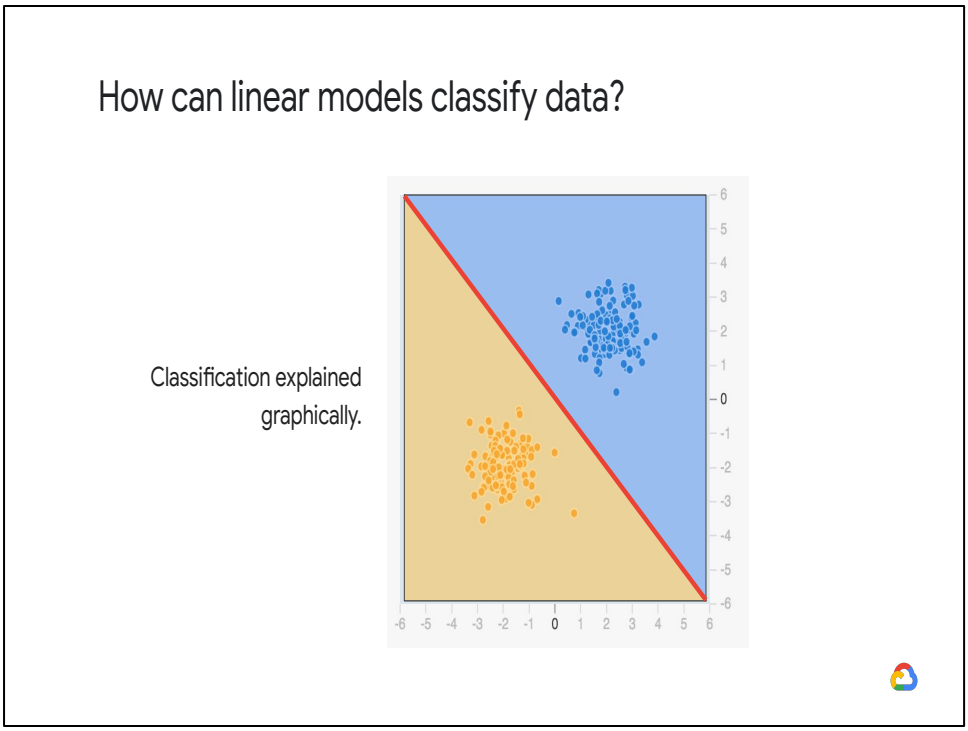
Linear models were one of the first sorts of ML models. They remain an important and widely used class of model today though.

In a linear model, small changes in the independent variables, or *features* as referred to in machine learning, yield the same amount of change in the dependent variable, or *label*, regardless of where the change takes place in the input-space. Visually, this looks like a line in 2D space, and the formula used to model the relationship is simply y = mx + b, where *m* captures the amount of change you observe in your label in response to a small change in your feature.

This same concept of a relationship defined by a fixed ratio of change between labels and features can be extended to arbitrarily high dimensionality, both with respect to the inputs and the outputs, meaning you can build models that accept many more features as input, model multiple labels simultaneously, or *both*. When you increase the dimensionality of the input, your slope term m must become n-dimensional. This new term is called the *weight*.

Visually, this process yields the n-dimensional generalization of a line, called a hyperplane, which is depicted on the right-hand side. When the dimensionality of the outputs is increased, the y and c terms become vectors of dimensionality n too. The b term, whether as a scalar or a vector, is referred to the bias term.

https://www.codecogs.com/latex/eqneditor.php
y = \beta +w^{T}\times X



How a linear model can be used for regression should be somewhat intuitive: you simply use the formula b + m times x to get your prediction, y. But how can a linear model be used for classification? How do you take a continuous number and interpret it as a class?

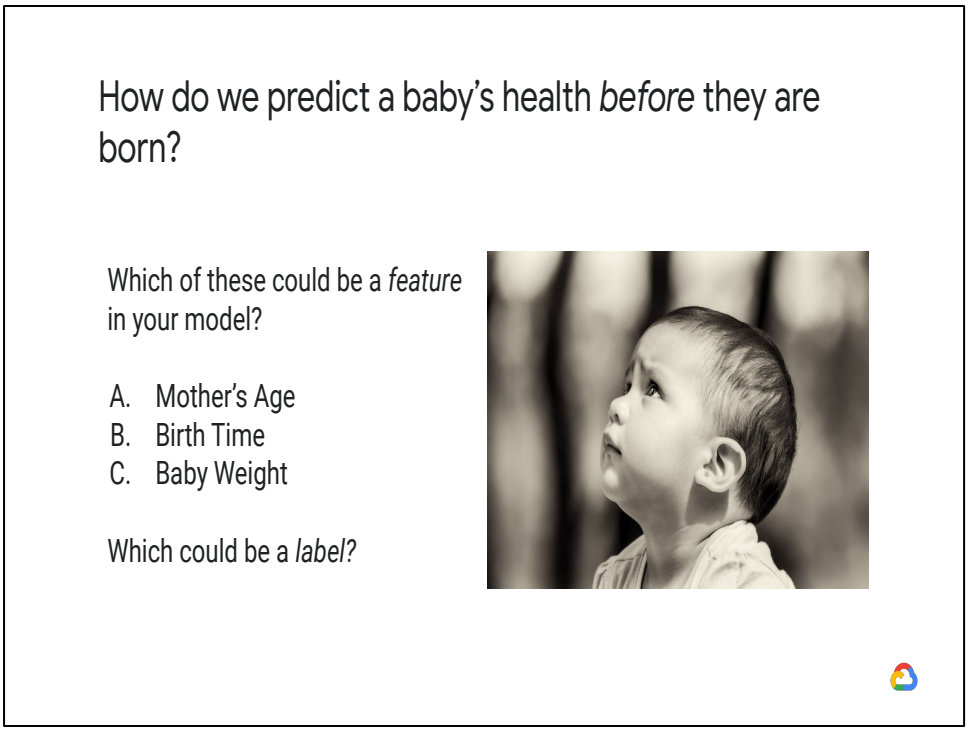
In order to take your model's numerical output and turn it into a class, you need to first think about how class membership can be encoded. The simplest way to encode class membership is with a binary: either you're a member or you're not. Of course, in many cases, categorical variables can take more than two values. This approach still works though! Just pretend that each value is its own independent class. For now though, stick with a single binary class.

Once you adopt this representation of the label, the task is more manageable. Now, you need a way to map your line onto a binary classification rule. One easy way to do this is to simply rely on the sign of the output. Graphically, that involves dividing your graph into two regions: the points above the line and the points below the line.

This line is called the decision boundary, because it reflects your decision about where the classes begin and end. Critically, the decision boundary is intended not just to be descriptive of the current data, it's intended to be predictive of unseen data. This property of extending to unseen examples is called "generalization," and it's essential to ML models.

Now, imagine rotating this figure until you were even with the red plane. What would

you see?



Babies are precious, but not all babies get the care they need. Some of them need urgent care immediately after they're born. The sorts of doctors who provide such care are scarce. In a perfect world, we'd know precisely where to send doctors so that the babies who need them get the care they need. But we don't live in that world.

How might predicting a baby's health before they are born be an ML problem? Well, if you knew which babies needed care before they were born, you could make sure you had doctors on-hand to care for them.

Assuming you want to make predictions before the baby is born, which of these could be a *feature* in your model?
Mother's Age
Birth Time
Baby Weight

Assuming you want to make predictions before the baby is born, which of these could be a *label* in your model?
Mother's Age
Birth Time
Baby Weight

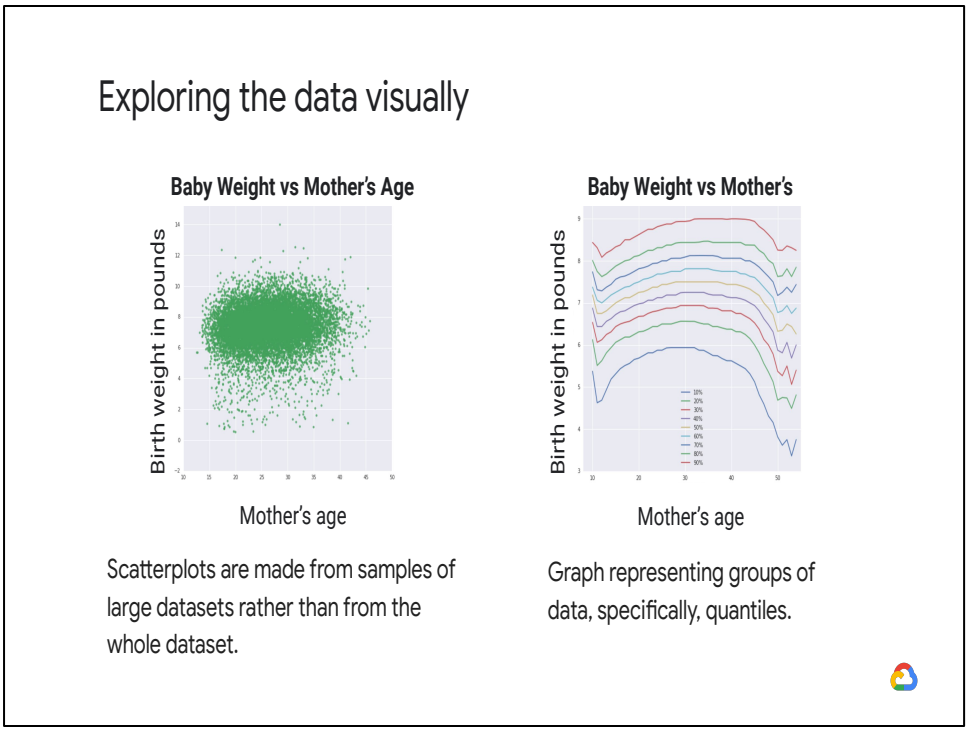
Answers:
(1) B
(2) C

If you didn't know the answer these questions, that's okay, because a lot of this is quite domain-specific. What you should have intuitions about, however, are when the information is available relative to when you want to make predictions. In this case, Birth Time is not available to you until well after the baby is born and so you can't use it.

Baby weight also happens to be an important indicator of baby health. Mother's age is something you can observe and which is predictive of baby weight. So, this seems like a good ML problem because there is a demonstrated need to know something that is too expensive to wait for (baby health) and which seems to be predictable beforehand.

Follow up question: assuming that you've chosen baby weight as your label, what sort of ML problem is this? Hint: baby weight is a continuous numeric value like 6.2 pounds.

Image cc0 (baby):
https://pixabay.com/en/kid-baby-boy-blur-black-and-white-2606329/



For now, treat this as a regression problem, and to simplify things, consider only the feature Mother's Age and the label Baby Weight. This data comes from a dataset collected by the US government. This healthcare dataset -- called the natality dataset (natality means birth) -- is available as a public dataset in BigQuery.

Often, the first step to modeling data is to look at the data, to verify that there is some signal and that it's not all noise. On the left, Baby Weight has been graphed as a function of Mother's Age using a scatterplot. Usually, scatterplots are made from samples of large datasets rather than from the whole dataset. Why use samples? Firstly, because scatterplotting too much data is computationally infeasible, and scatterplots with lots of data become visually hard to interpret. Note that there seems to be a small positive relationship between mother's age and baby weight.

On the right is a plot that uses the same two variables but, unlike a scatterplot, which represents data points individually, *this* graph represents *groups of data*, specifically, quantiles. As a result, we needn't sample before building it and there's consequently no risk of it getting a non-representative sample. As an added bonus, the results are also repeatable and parallelizable. This plot, which is about 22 gigabytes of data, was produced in just a few seconds using BigQuery.

Do you see any relationship in the data just by looking at it?

You might have noticed something that wasn't apparent on your scatterplot: baby weight seems to reach its maximum when mothers are around 30, and it tapers off as

mothers get both older and younger. This suggests a non-linear relationship, which is both something that wasn't apparent with your scatterplot and an ominous sign, given the intention to model this relationship with a linear model. In fact, the intention to model a non-linear function with a linear model is an example of what's called "underfitting". You might wonder why a more complex type of model is being used. Here, it's for pedagogical reasons. Model selection, and a concept known as "overfitting", will be discussed later. In short: there are risks that are proportional to model complexity.

Equation for a linear model tying mother's age and baby weight

The slope of the line is given by w1.

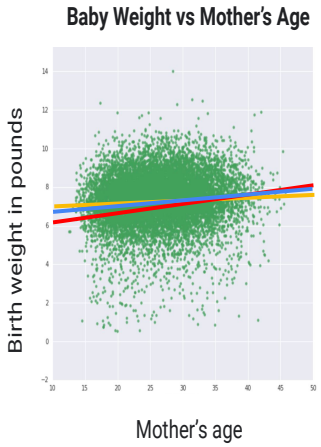
$$y = w_1 x_1 + b$$

- x_1 is the **feature** (e.g. mother's age)
- w_1 is the **weight** for x_1

`Line: y= .02x + 6.83`

`Line: y= .03x + 6.49`

`Line: y= .01x + 7.14`

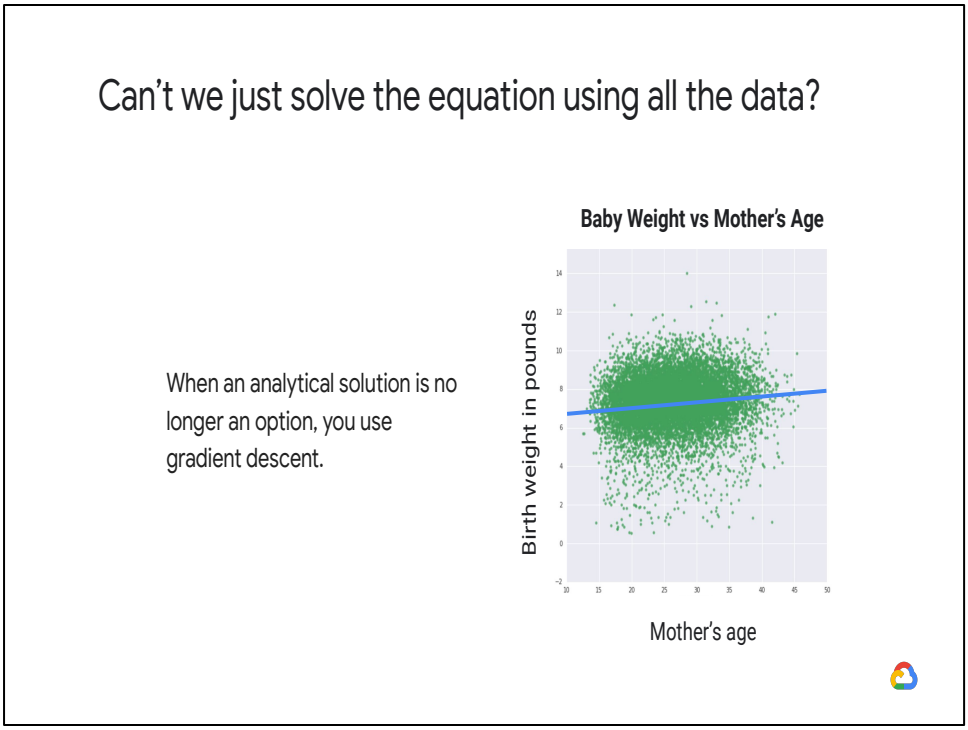


It appears that there's a slight positive relationship between Mother's Age and Baby Weight. You're going to model this with a line.

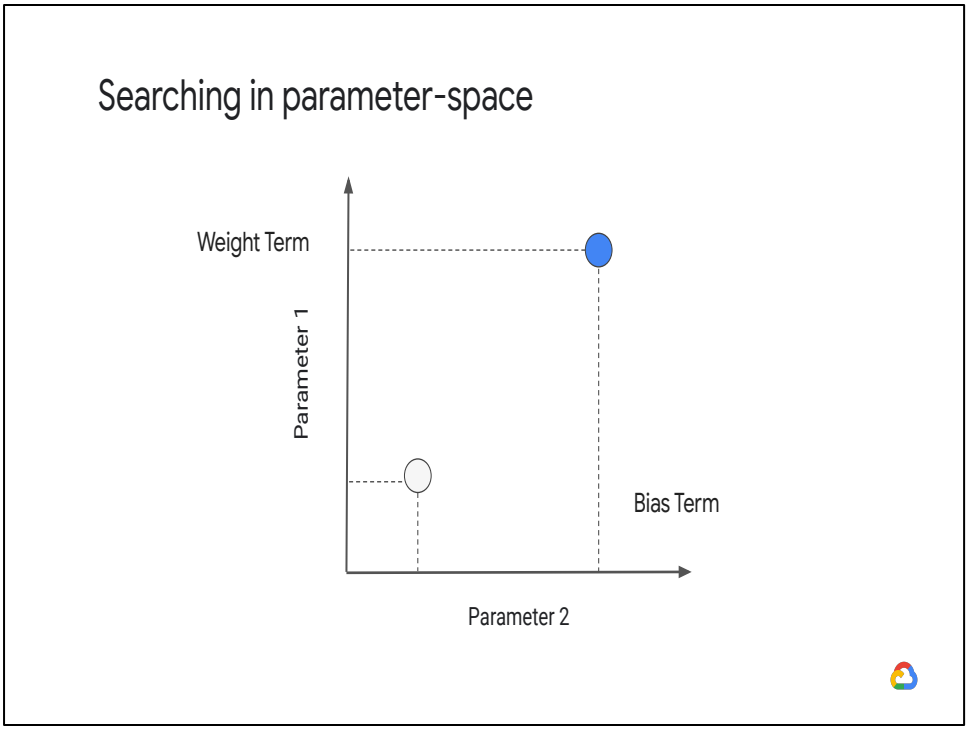
Given that you're using a linear model, your earlier intuition translates into an upward sloping blue line with a positive y-intercept.

You've eyeballed the data to select this red line, but how do you know whether that line should be higher or lower by adjusting the weight? How do you know it's in the right place?

How, for example, do you know it's actually better than the yellow line?

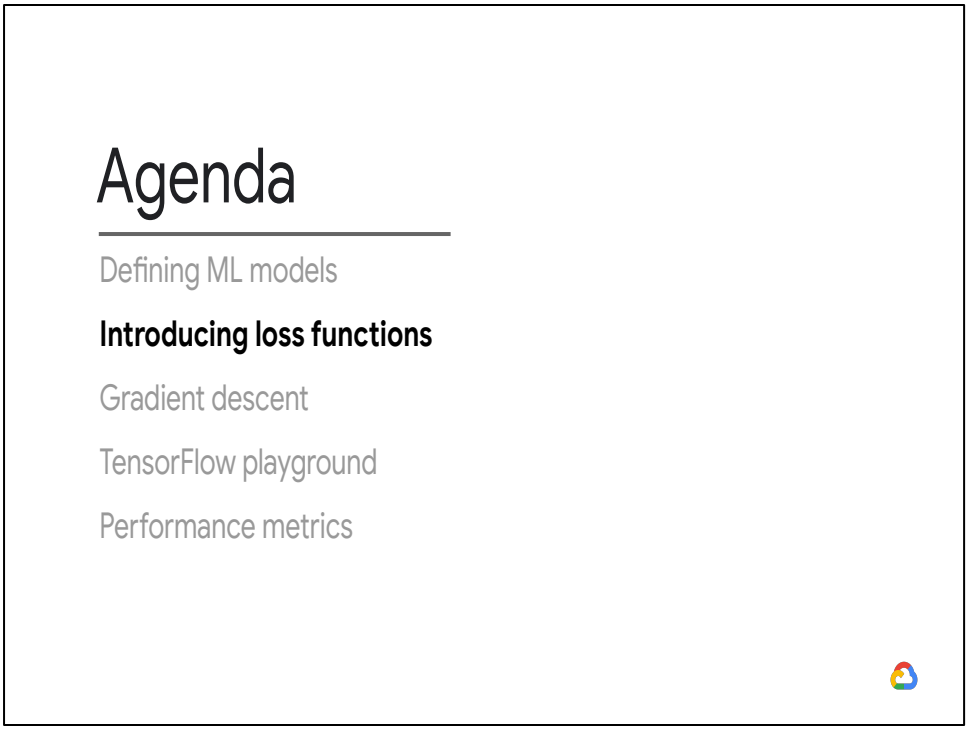


If you've taken Statistics, you may remember seeing a process for determining the best weights called *least squares regression*. It's true that there are ways of analytically determining the best possible weights for linear models. The problem is that these solutions only work up to a certain scale. Once you start using really big data sets, the computation required to analytically solve this problem becomes impractical. What do you do when an analytical solution is no longer an option? You use gradient descent.



Start by thinking about optimization as a search in parameter-space. Remember that your simple linear model has two parameters, a weight term and a bias term. Because they are both real-valued, you can think of the space of all combinations of values for these two parameters as points in 2D space. But remember, you're looking for the best value.

How does one point in parameter-space vary from another with respect to quality? First, the question needs to be reframed a little. Because input-spaces, which are the spaces where data live, are often themselves infinite, it's not possible to evaluate the parameters on every point in input space. So, you estimate what this calculation would look like using what you have: your training data. And to do that, you'll need to somehow generalize from the quality of a prediction for a single data point, which is simply the error of that prediction, to a number that captures the quality of a group of predictions. The functions that do this are known as loss functions.



In the previous topic, models were defined as mathematical functions using parameters and hyperparameters, and you were introduced to the parameters for linear models. You learned how analytical methods for finding the best set of model parameters don't scale, and how you can think of optimizing your parameters as searching through parameter-space. But to compare one point to another, you will need some sort of measure.

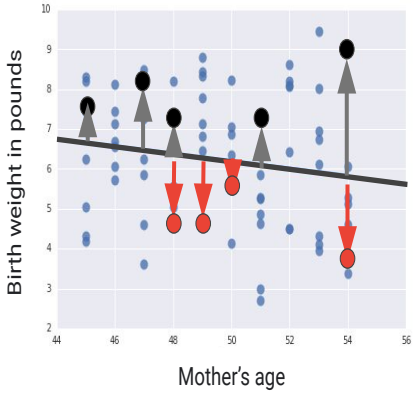
In this topic, you'll learn about loss functions, which are able to take the quality of predictions for a group of data points from your training set and compose them into a single number with which to estimate the quality of the model's current parameters.

Compose a loss function by calculating errors

Error = actual (true) - predicted value
Compute the errors:

```
+0.70
+1.10
+0.65
-1.20
-1.15
+1.10
+3.09
-2.10
```

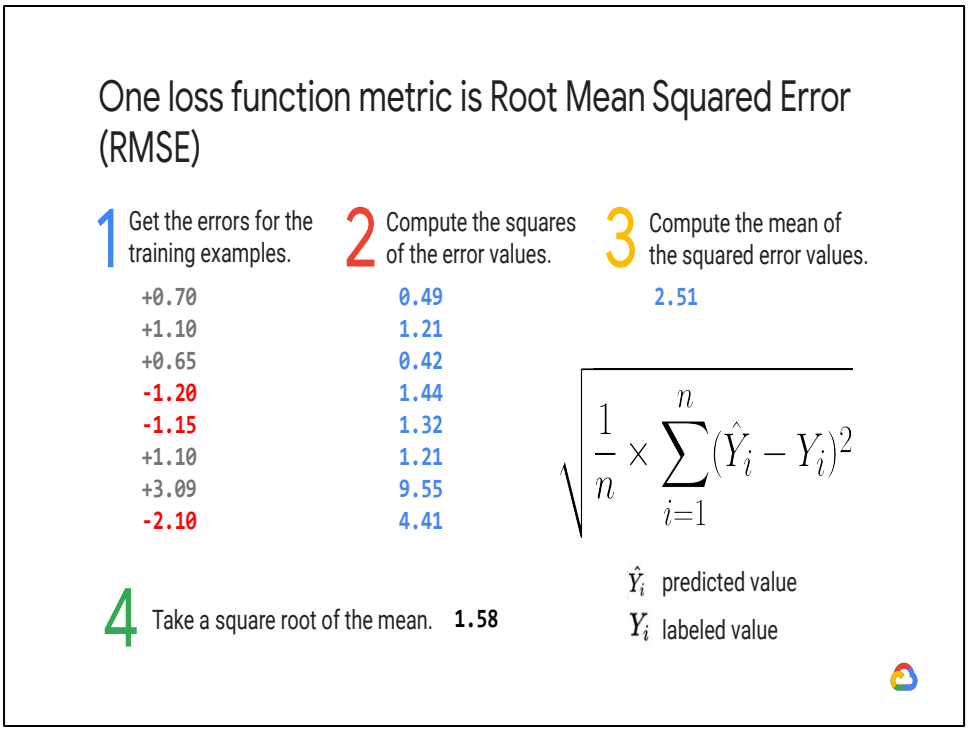
Each error makes sense. How about all the errors added together?



One measure of the quality of the prediction at a single point in your data set is simply the signed difference between the prediction and the actual value, or *label*. This difference is called the error.

How might you put a bunch of error values together? The simplest way to compose them is a SUM.

If you were to use the sum function to compose your error terms, the resulting model would treat error terms of opposite sign as cancelling each other out. And while your model *does* need to cope with contradictory evidence, it's not the case that a model that splits the difference between positive and negative errors has found a perfect solution. You'd like to reserve the "perfect" designation for a model in which the predictions match the label for all points in your dataset, not for a model that makes signed errors that cancel each other out.

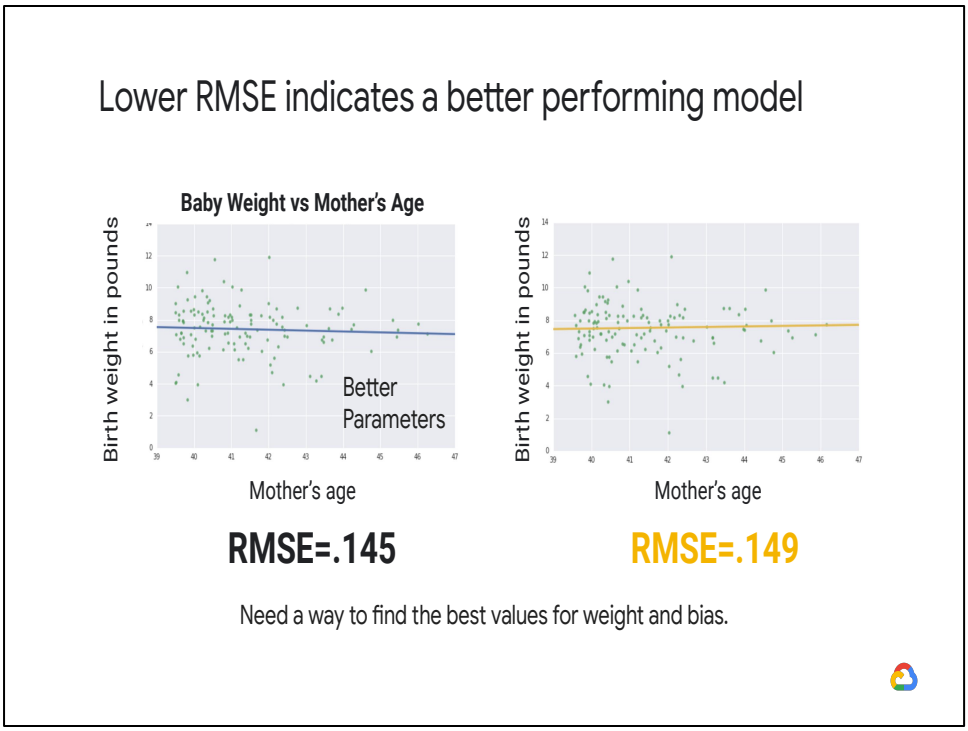


The sum of the absolute values of the errors seems like a reasonable alternative, but there are problems with this method of composing data as well, which will be tackled shortly.
Instead, what is often used is what is called the Mean Squared Error. The MSE is computed by taking the set of errors from your dataset, taking their squares (to get rid of the negatives), and computing the average of the squares.

The MSE is a perfectly valid loss function, but it has one problem. Although errors might be in pounds, or kilometers, or dollars, the square error will be pounds-squared, kilometers-squared, or dollars-squared. That can make the MSE hard to interpret.
As a result, it is common practice to take the square-root of that mean squared error to get to units that can be understood. RMSE is the root of the mean squared error.

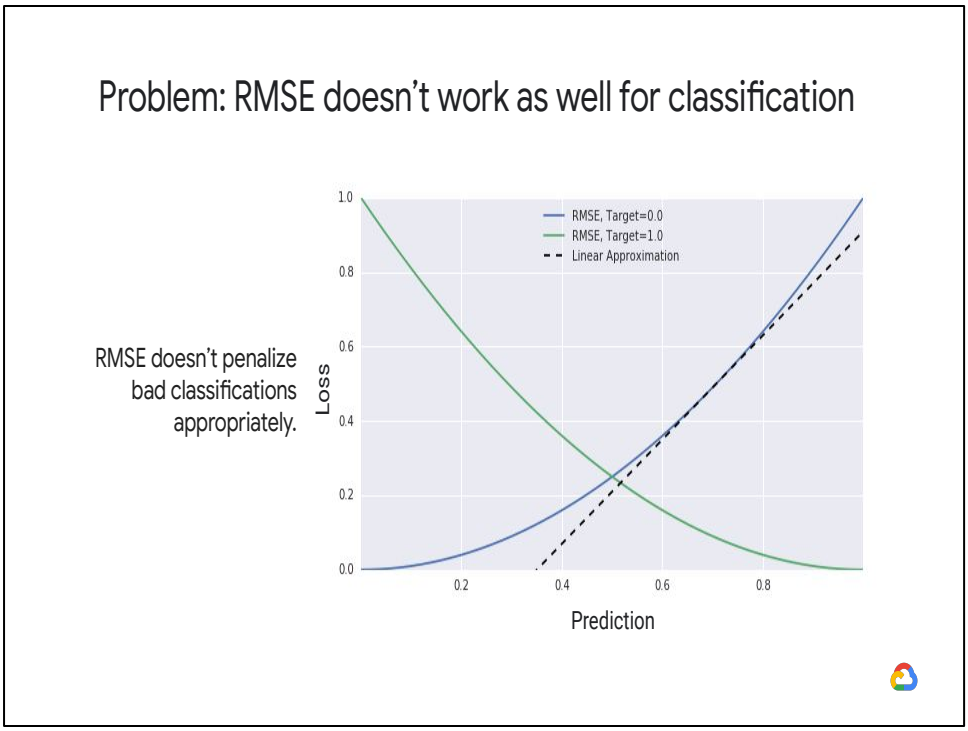
The bigger the RMSE, the worse the quality of the predictions. So, what you want to do is to minimize the RMSE.
The notation here is to use a little hat symbol on top of the Y that represents your model's prediction, and to use a plain Y to represent the label.

https://www.codecogs.com/eqnedit.php?latex=sqrt{\frac{1}{n}&space;\times&space;\sum_{i=1}^{n}(\hat{Y}_{i}-Y_{i})^{2}}



Now you have a metric to compare two points in parameter-space--two sets of parameter values for your linear model--formally.
Take a look at these two scatterplots and regression lines for Baby Weight vs Mother's Age for mothers above 39. It can be incredibly hard to visually spot which line is a better fit to the underlying data. That's where your loss metrics aid in deciding which model is better.

The model on the left has an RMSE of .145, and the model on the right has an RMSE of .149. Therefore, the loss function indicates that the values for weight and bias on the left-hand side are better than on the right-hand side.



Although RMSE works fine for linear regression problems, it doesn't work as a loss function for classification. Remember, classification problems are ones in which the label is a categorical variable. The problem with using RMSE for classification has to do with how these categorical variables are represented in your model. As discussed, categorical variables are often represented as binary integers.

For an intuition as to why this presents a problem, look at the loss curves depicted. The domain, on the X axis, represents the prediction. The range, on the Y axis, represents the loss, given that prediction. Color here denotes the label: green indicates the label was 1, blue indicates the label was 0.

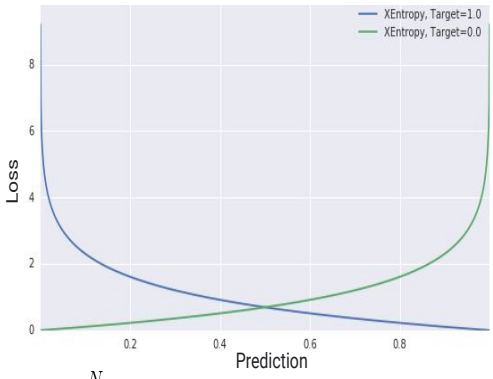
What's wrong with this curve?

The problem is, it fails to capture an intuitive belief that predictions that are really bad should be penalized much more strongly. Note how a prediction of 1 when the target is 0 is about 3 times worse than a prediction of 0.5 for the same target.

Instead of RMSE then, you need a new loss function, one that penalizes in accordance to your intuitions.

Problem: RMSE doesn't work as well for classification

Bad classifications are penalized appropriately.



$$\frac{-1}{N} \times \sum_{1}^{N} y_i \times log(\hat{y}_i) + (1 - y_i) \times log(1 - \hat{y}_i)$$

One of the most commonly used loss functions for classification is called Cross-Entropy or Log Loss. Here you have similar graph to what you saw on the last slide, only instead of showing the loss for RMSE, the value of a new loss function, called Cross-Entropy, is shown. Note that unlike RMSE, Cross-Entropy penalizes bad predictions very strongly, even in this limited domain.

Computing cross-entropy loss

Positive term Negative term

$$\frac{-1}{N} \times \sum_{1}^{N} y_i \times log(\hat{y}_i) + (1 - y_i) \times log(1 - \hat{y}_i)$$

X	Y_i	$\hat{Y}_i$
	1	.7
	0	.2

(1.0*log(.7) + ~~(1-1.0)*log(1-.7)~~

+

~~(0.0*log(.2)~~ + (1-0.0)*log(1-.2))

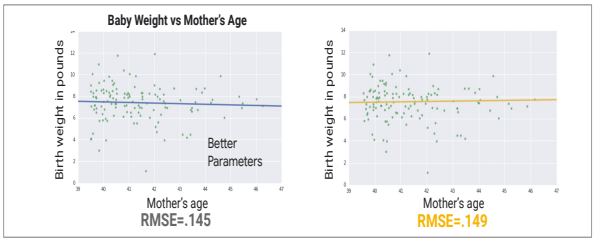
*(-½)=.13

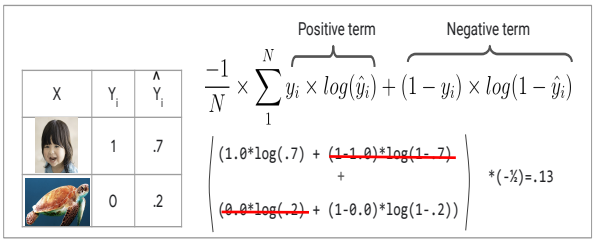
The formula for Cross Entropy boils down to two different terms, only one of which will participate in the loss for a given data point. The first term participates for “positive” examples, where the label, Yi, is 1. The second term participates when the label is 0.

This table shows both the labels as well as the predictions for two pictures in an image classification task. The label encodes whether the picture depicts a human face. The model seems to be doing a decent job: the prediction is much higher for the example on the top as compared with the example on the bottom.

The way the loss function is constructed, the negative term from the first example and the positive term from the second example both drop out. Given predictions of .7 and .2 for two data points with labels 1 and 0, the cross entropy loss is effectively the positive term for the first data point plus the negative term for the second point multiplied by negative ½. The result is .13.

From loss functions to gradient descent

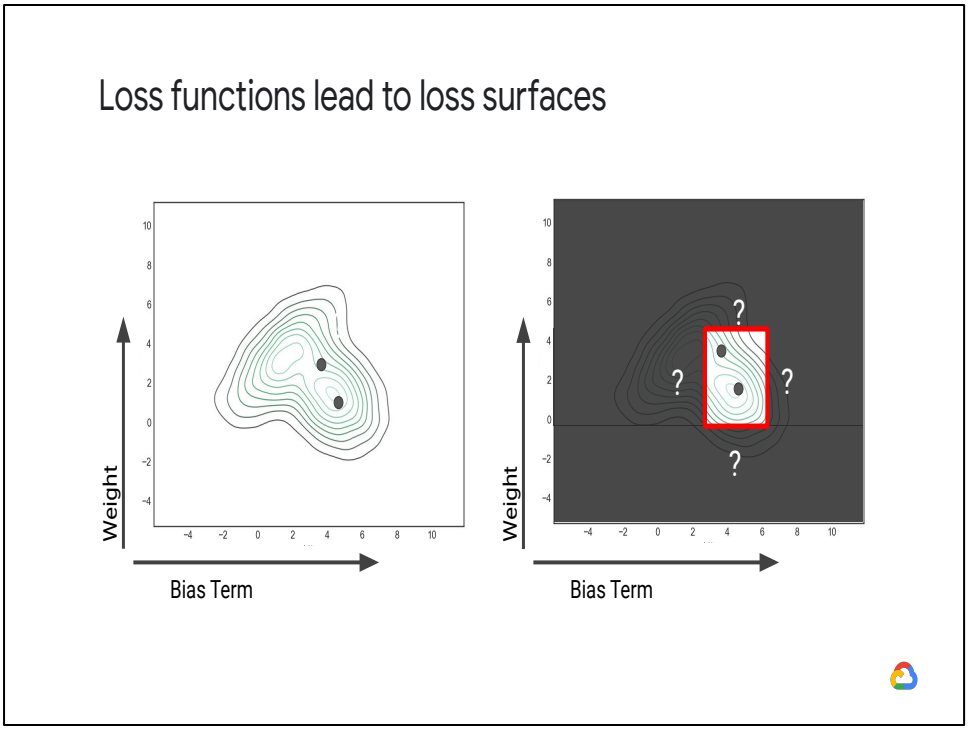




You now know how to compare two points in parameter-space, whether you're using RMSE for regression, or XEntropy for classification. But remember that your goal is to find the best set of parameters, or the best point in parameter-space. How do you use your knowledge of how to compare the two sets of parameters and turn it into a search strategy?

In the previous topic, optimization was framed as a search in parameter-space and then loss functions were introduced as a way to compare these points.

How do you take a loss function and turn it into a search strategy? That's where gradient descent comes in. Gradient descent refers to the process of walking down the surface formed by using your loss function in parameter-space.



That surface might actually look like this on the left. Of course, this is what it would look like with perfect information, i.e. with complete knowledge of the space.

In actuality, you'll only know loss values at the points in parameter-space where you've evaluated your loss function -- or just the two points in the red-bounded area shown on the right. However, you will somehow need to make a decision of where to go next to find the minimum anyway.

Image cc0 (topographic map):
https://pixabay.com/en/maps-topographical-maps-geography-2294276/

Finding the bottom

Which direction should I head?

How large or small a step?

It turns out that the problem of finding the bottom can be decomposed into two different and important questions:

(1) Which direction should I head?
(2) How far away should I step?

Make a simplifying assumption for the moment and use a fixed-size step.

Image cc0 (compass):
https://pixabay.com/en/adventure-compass-hand-macro-1850673/
Image cc0 (rappel):
https://pixabay.com/en/climbing-rappelling-canyoneering-1761386/

A simple algorithm to find the minimum

```
while loss is > Epsilon:
    direction = computeDirection()
    for i in range(self.params):
        self.params[i] = //
            self.params[i] //
                + stepSize * direction[i]
    loss = computeLoss()
```

Epsilon = A tiny Constant

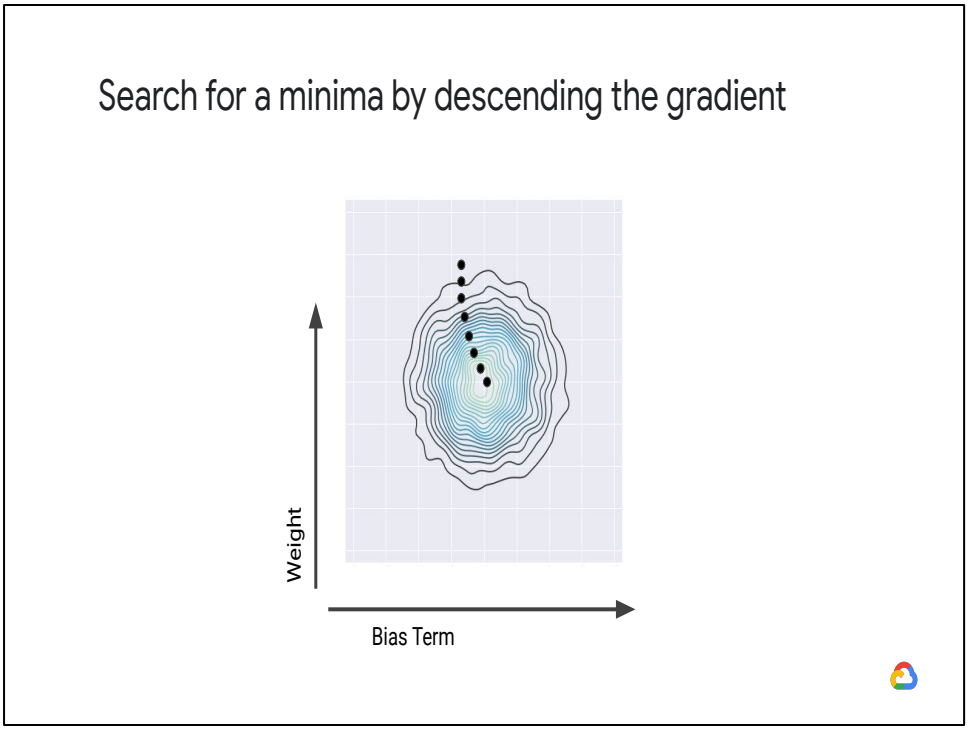
This lends itself to a very simple algorithm.

While the loss is greater than a tiny constant,

(1) Compute the direction

(2) For each parameter in the model, set its value to be the old value plus the product of the step size and the direction

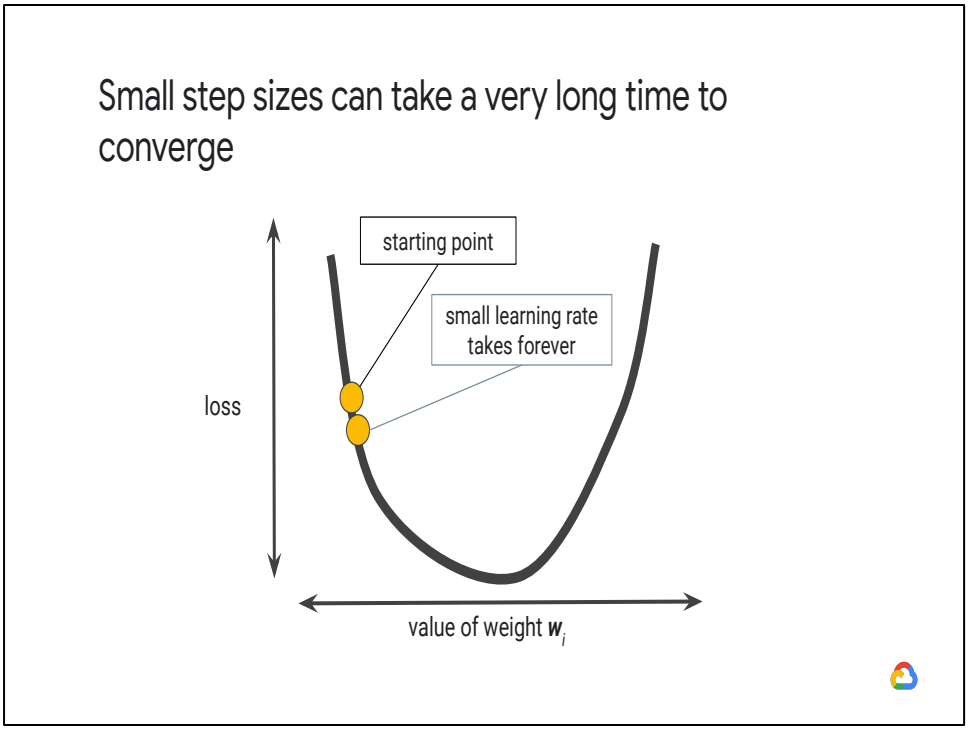
(3) Recompute the loss



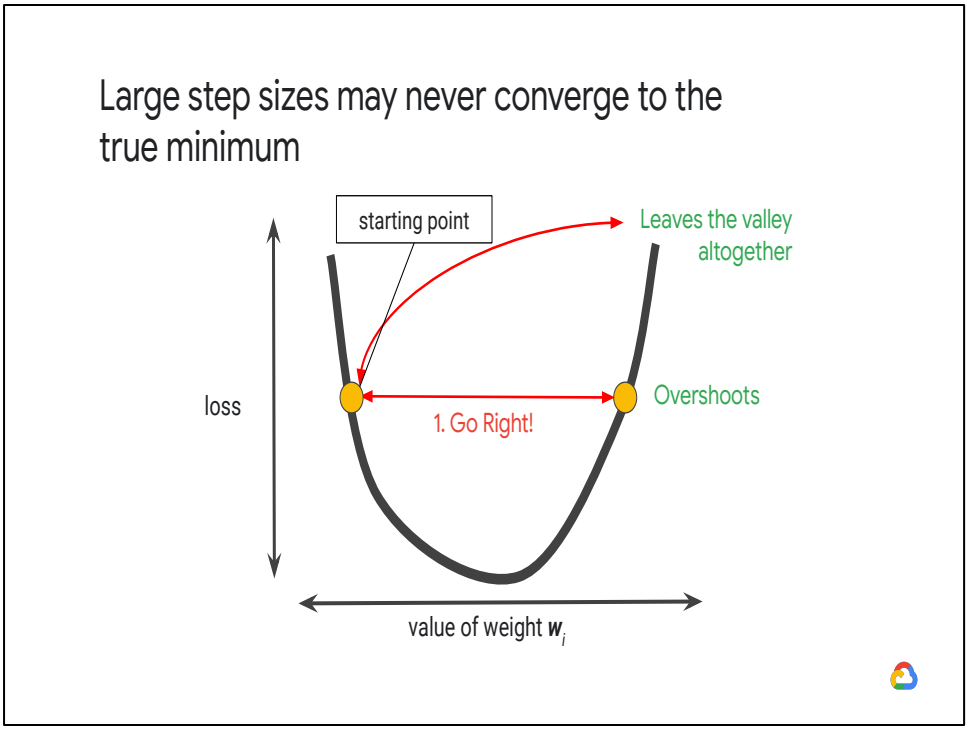
You can think of a loss surface as a topographic or contour map. Every line represents a specific depth. The closer the lines are together, the steeper the surface is at that point.

The algorithm takes steps, which are represented here as dots. In this case, the algorithm started on the top edge and worked its way down toward the minimum.

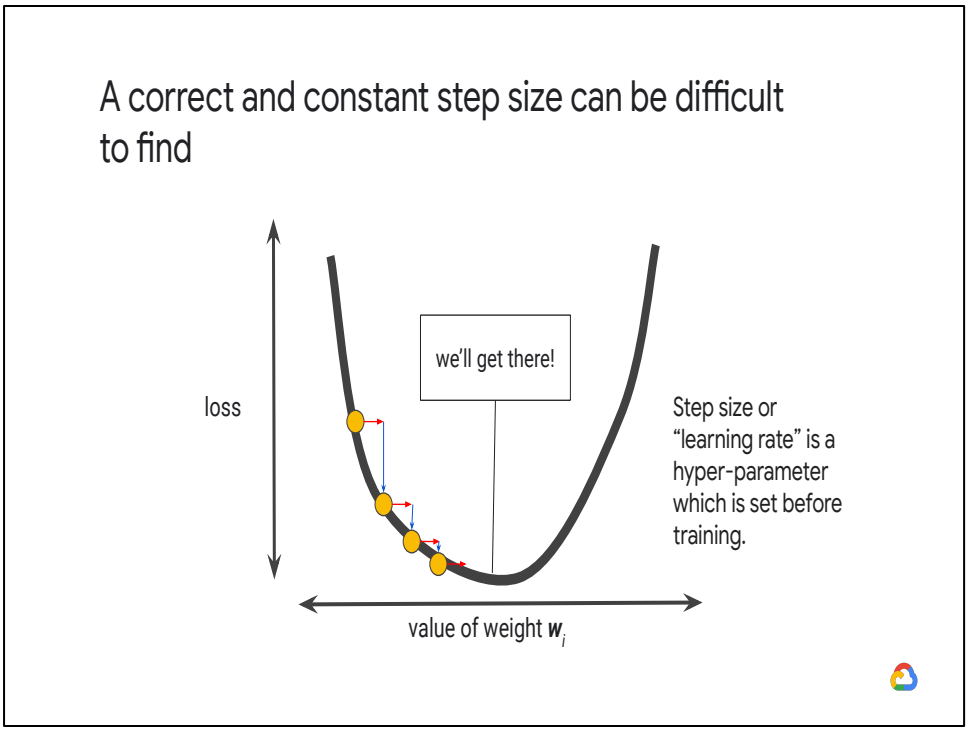
Note how the algorithm takes fixed-size steps in the direction of the minimum.



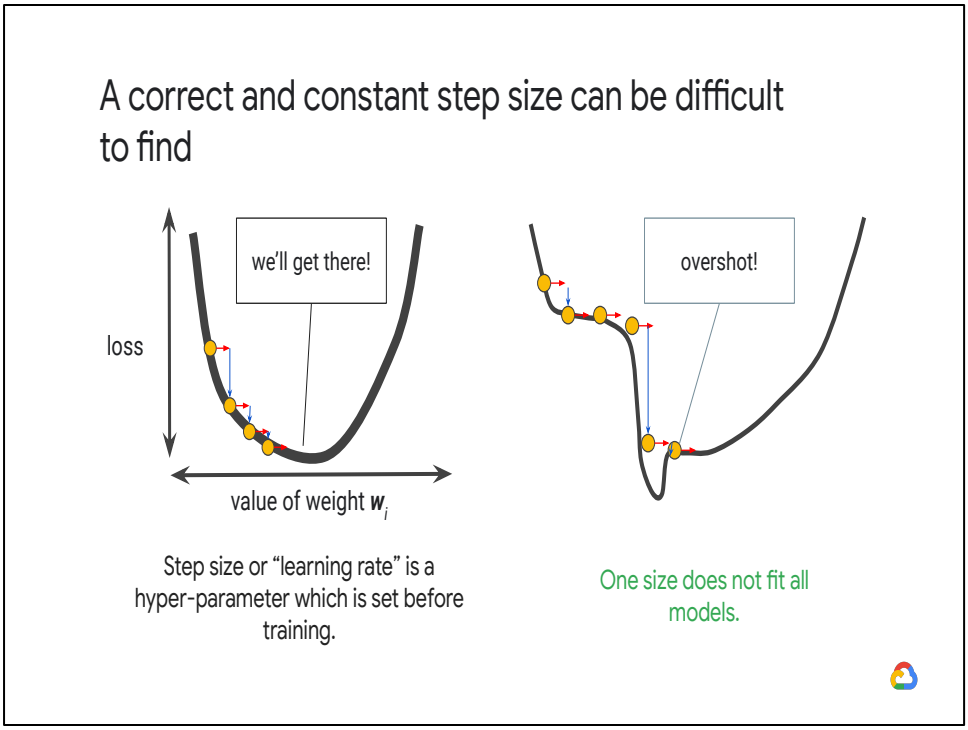
Putting aside direction for the moment, if your step size is too small, your training might take forever. You're guaranteed to find the minimum though (so long as there's only one minimum like this linear regression loss curve here -- remember your two minimum loss surface just shown previously? How to deal with those efficiently will be covered later).



If your step size is too big, you might either bounce from wall to wall, or bounce out of your valley entirely, and into an entirely new part of the losos surface. Because of this, when the step size is too big, the process is not guaranteed to converge.

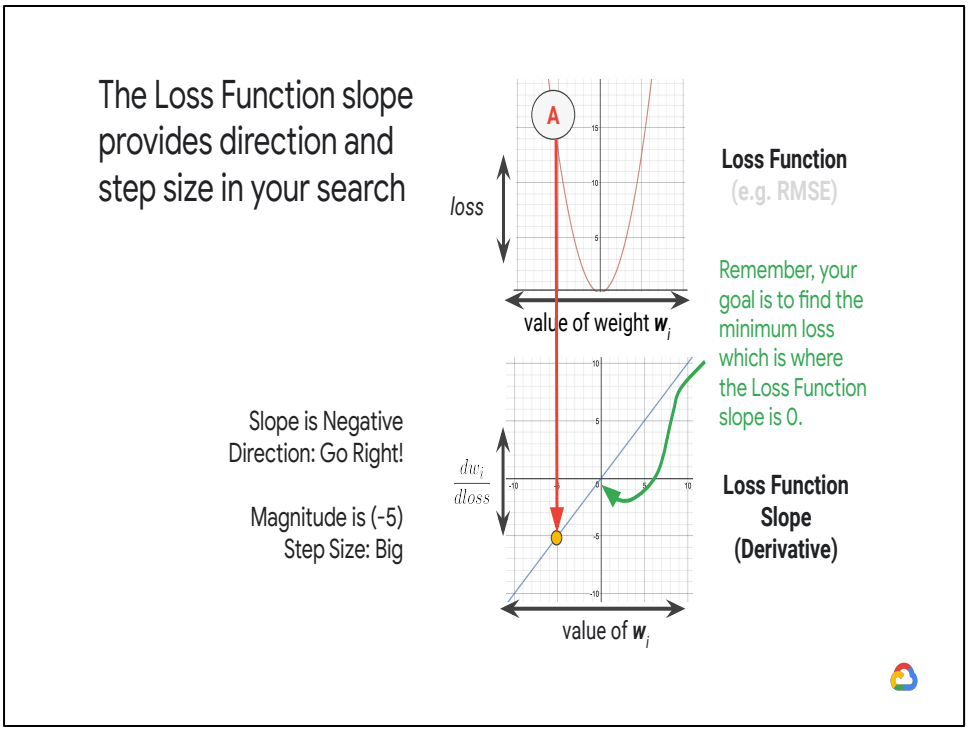


If your step size is just right, then you're set. But whatever this value is, it's unlikely to be just as good on a different problem.

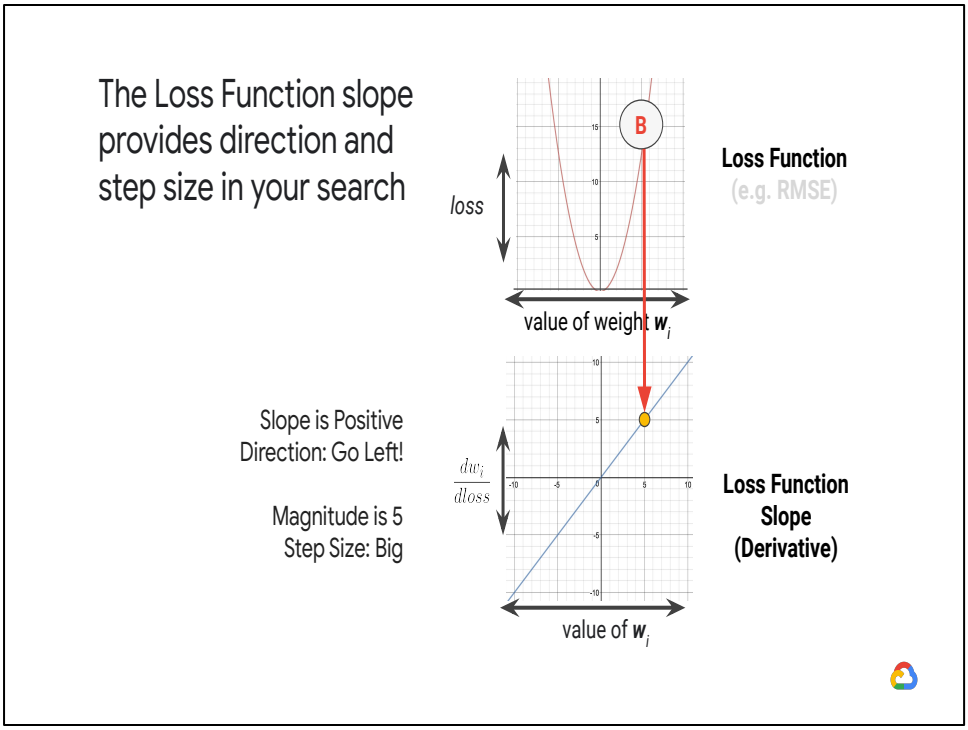


Note that the step size, which seemed to work on the left-hand curve, fails miserably on the right-hand curve.

One size does not fit all models. So, how should you vary step size?

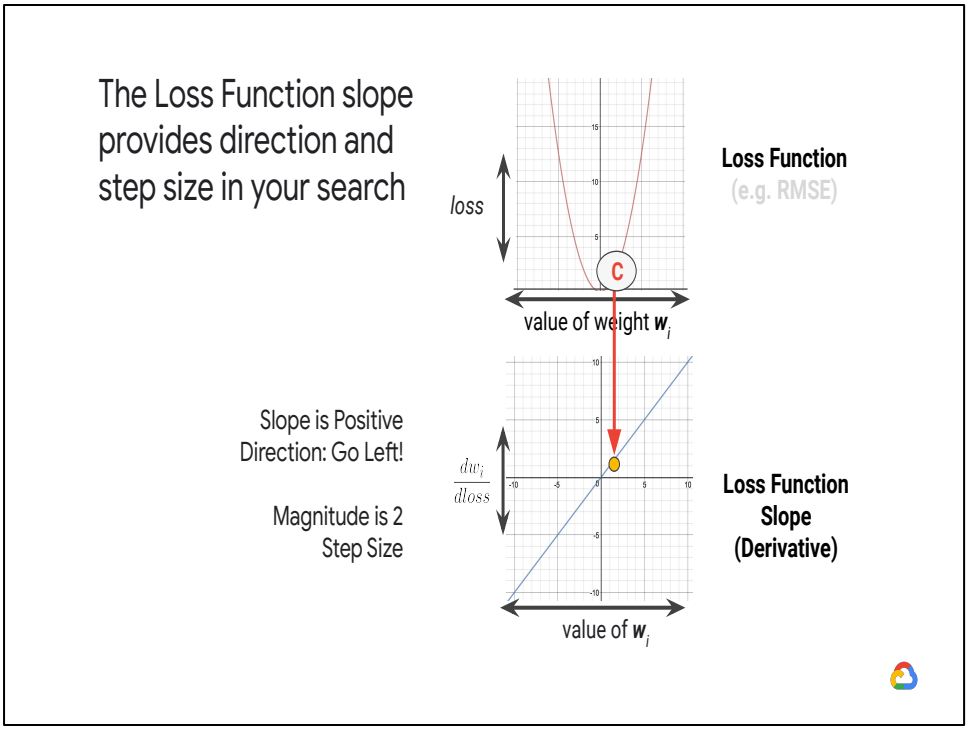


Thankfully, the slope (or the rate at which the curve is changing) gives us a decent sense of how far to step and the direction at the same time. Look at that bottom subplot showing the value of the slope at various points along the weight-loss curve. Note that where the values are bigger, we are generally farther away from the bottom than where the slope is small. Note also that where the slope is negative, the bottom on the top chart is to the right and where the slope is positive, the bottom on the top chart is to the left.



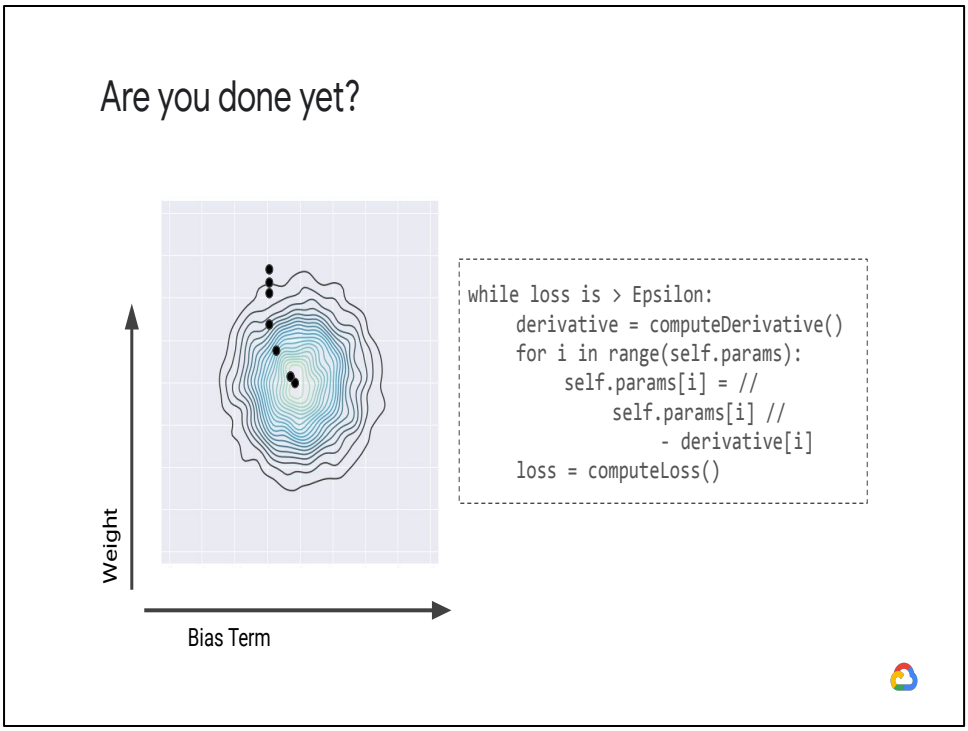
Here's another example. Look at point B. Does it have a positive or negative slope?

It has a positive slope, which tells you to go left to find a minima. Note that the slope is steep, which means you take a big step.



You're getting closer now. Take a look at point C on the loss surface. Does it have a positive or negative slope and does it have a steep slope?

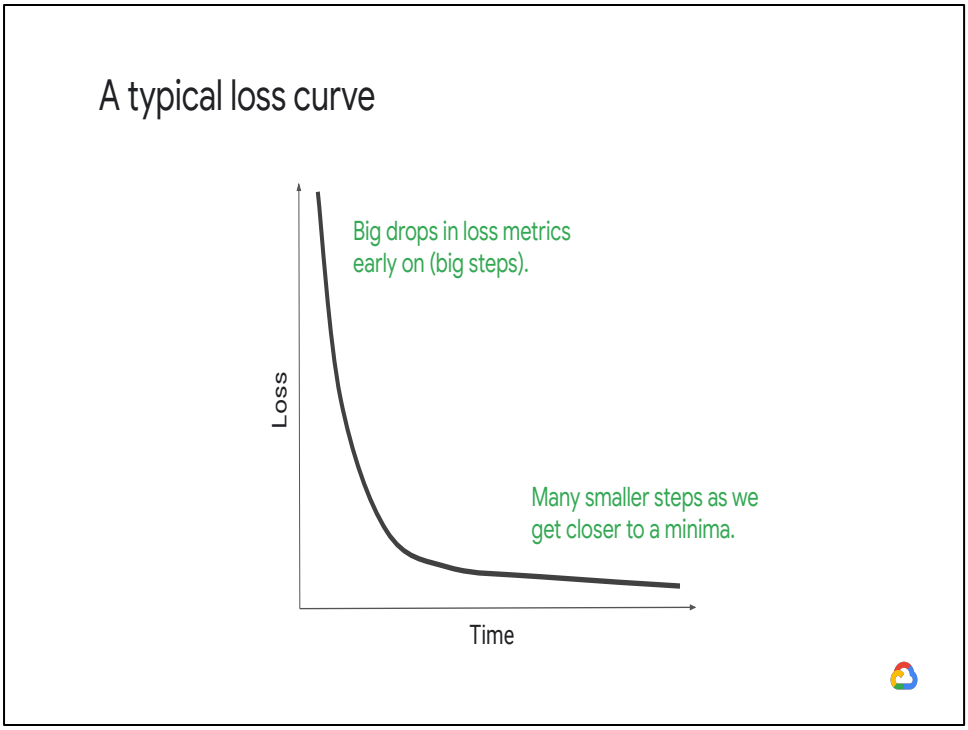
It's a positive slope again, which means still travel left. Here the slope is much more gradual so you're going to take smaller steps so you don't accidentally step over a minima.



Now you've replaced your two calls to computeDirection() and computeStepSize with a single call to your new function, computeDerivative(). You've also updated your loop for updating the model's parameters, setting each parameter to be its old value minus the partial derivative of that parameter with respect to the loss.

Are you done yet? You seem to have found a way to take steps in the right direction with the appropriate size. What could go wrong? Well, empirical performance.

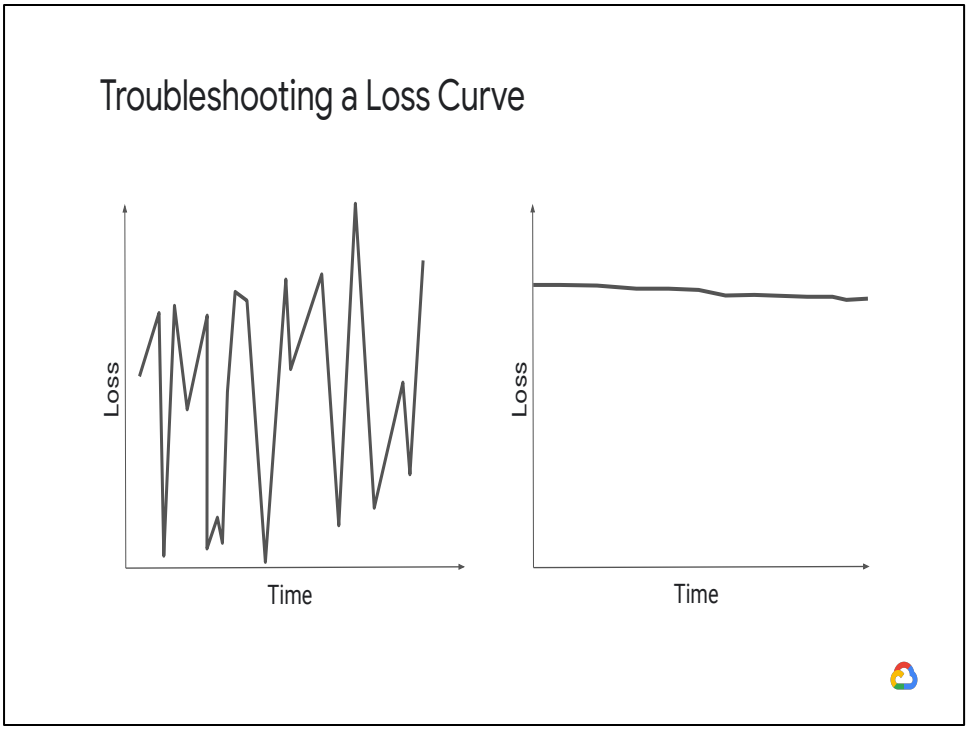
It turns out that with respect to the set of problems that ML researchers have worked on, which is to say, the set of loss surfaces in which you've applied this procedure, your basic algorithm often either takes too long, finds sub-optimal minima, or doesn't finish. To be clear, this doesn't mean that your algorithm doesn't work. It simply means you tend not to encounter the sorts of problems where it excels.



Before we go into one of the first ways that researchers addressed this problem, let's put some of the things you've learned together. Put yourself into the shoes of your model and look at how loss might change over time during training.

Imagine that you're performing gradient descent and updating your model's parameters with respect to the derivative of a loss function and you've configured things so that you can see how your loss is changing during training. This is a common scenario in machine learning, particularly when model training comprises hours or possibly even days; you can imagine how important it is not to waste days of time. So with that in mind, let's troubleshoot a loss curve.

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.



What if you see a loss curve like the one on the left? Assume for a moment that the scale of the loss axis is large. What does this tell you about your model and the way your search is going on the loss surface? What it means is that your search is jumping all around and not, as you'd like, making steady progress toward a particular minima.

What about the loss curve on the right? This one means you're probably still in the same valley, but it'll take a very, very long time to reach the bottom.

In both of these cases, the step size wasn't correct for that particular problem. In the first case, the step size was too big. In the second it was too small.

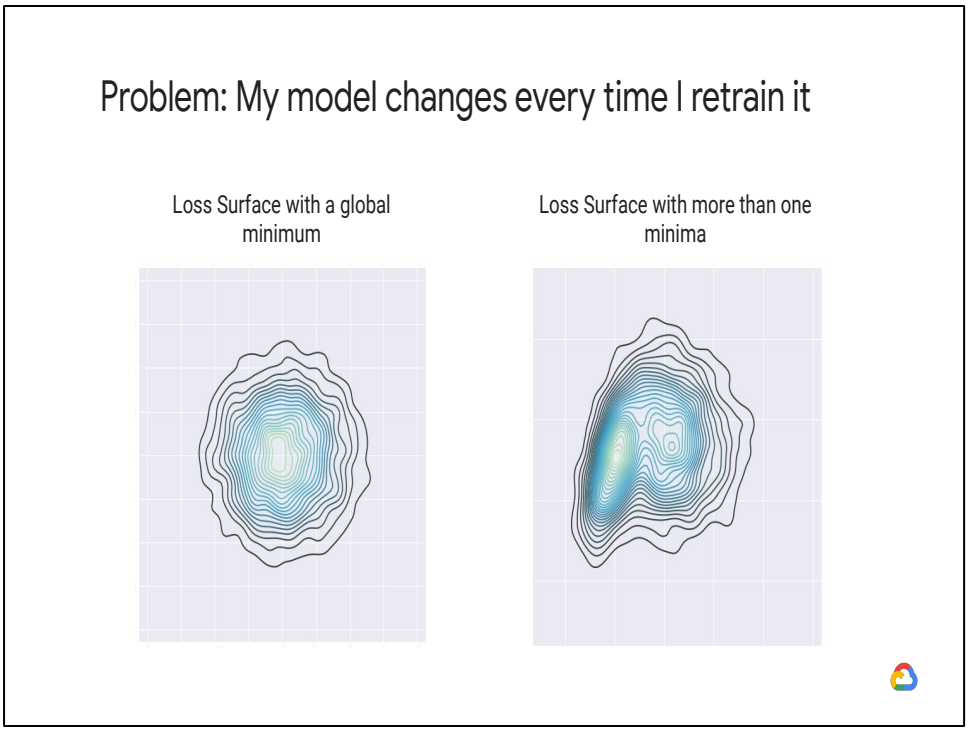
Adding a scaling hyperparameter

```
while loss is > Epsilon:
    derivative = computeDerivative()
    for i in range(self.params):
        self.params[i] = //
            self.params[i] //
                - learning_rate //
                * derivative[i]
    loss = computeLoss()
```

What you need is a scaling parameter. In the literature, this is referred to as the “learning rate” and with its introduction into your formula, you now have classic gradient descent.

You could imagine using brute force to figure out the best value scale for learning rate. But recall that learning rate is likely to have a problem-specific best value. Because it’s set before learning begins, learning rate is a hyper-parameter. And to determine the best value for hyper-parameters, there’s a better method available than brute force, which is called hyperparameter tuning. How to do this in Cloud ML Engine will be reviewed in a later module. Generally, learning rate is a fraction significantly less than one.

For now, simply remember this formulation of gradient descent and that learning rate is a hyperparameter that is fixed during gradient descent. In the final module you’ll get to see models performing gradient descent in real-time, and be able to change things like learning rate while observing a loss curve.



A common situation that practitioners encounter is that they re-run model code that they've written, expecting it to produce the same output. Only it doesn't.

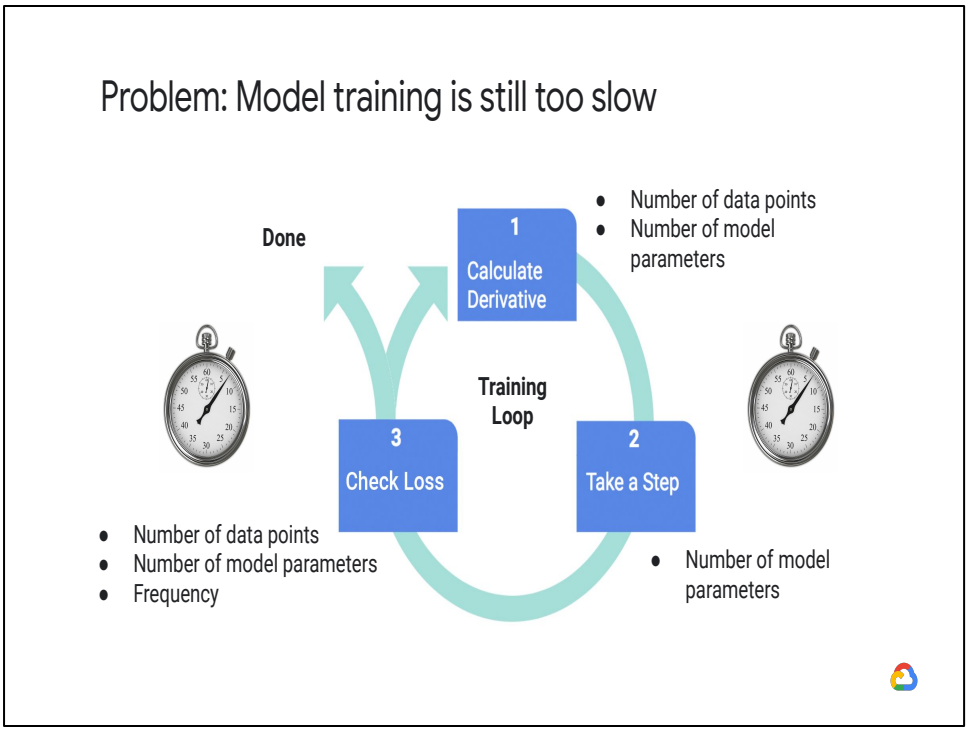
Programmers are often used to working in deterministic settings. In ML, this sometimes is not the case.

Although this will not happen if you're training a linear model, for many other models, if you re-train the model a second time, even when using the same hyper-parameter settings, the resulting parameter settings might be very different. This at first seems disconcerting. Aren't we looking for the best set of parameters? Does this mean that gradient descent isn't working, or that you've implemented it incorrectly?

Not necessarily. What it does mean is that instead of searching a loss surface like on the left-hand side, you are actually searching loss surfaces like on the right-hand side. Notice that whereas the left-hand side loss surface has a single bottom, the right-hand side has more than one. The formal name for this property is convexity: the left-hand side is convex whereas the right-hand side is non-convex.

Why might an ML model's loss surface have more than one minimum? Well, it means that there are a number of equivalent or close-to-equivalent points in parameter-space, meaning settings for your parameters that produce models with the same capacity to make predictions. This will be revisited when neural networks are introduced later on, because they're a prime example of where this happens, so it's okay if it's not clear how this could be the case. For now, simply keep in mind that loss

surfaces vary with respect to the number of minima they have.



Sometimes, fast just isn't fast enough. We all hate waiting for models to finish training. Is there any way to make training go even faster? Yes, but to understand what your options are, it's best to consider the high-level steps of your algorithm and their sources of time complexity.

When you calculate the derivative, the cost of the calculation is proportional to the number of data points you are putting into your loss function, as well as the number of parameters in your model. In practice, models can vary from tens of parameters to hundreds of millions. Similarly, datasets can vary from a few thousand points to hundreds of billions.

For the case of updating the model's parameters, this happens once per loop and its cost is determined solely by the number of parameters in the model. However, the cost of making the update is typically small relative to the other steps.

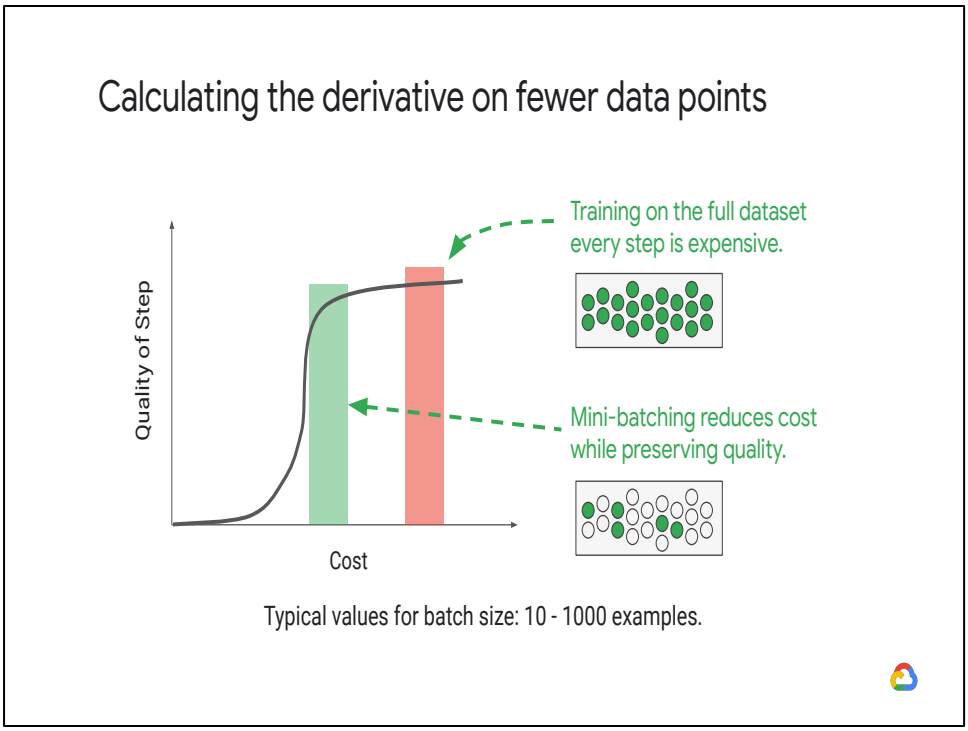
Finally, there's checking the loss. This step's time complexity is proportional to the number of data points in the set that you're using for measuring the loss and the complexity of your model. Surprisingly, even though you have represented this process as a loop, the Check Loss step needn't be done at every pass. The reason for this is that most changes in the loss function are incremental.

So, what can you change to improve training time? Typically, the number of effective parameters in a model is fixed, although how this might be varied will be returned to in a future module on regularization. Additionally, although it might sound appealing to

reduce the number of data points used to check the loss, this is generally not recommended as you should check against all available data.

Instead, you have two main knobs to turn to improve training time: the number of data points you calculate the derivative on, and the frequency with which you check the loss.

Image (timer) cc0:
https://pixabay.com/en/stopwatch-timer-clock-symbol-icon-2624277/



As discussed, one of the knobs you can turn to speed up model training is the number of data points that you calculate the derivative on. Remember, the derivative comes from your loss function and your loss function composes the error of a number of predictions together. So, this method essentially reduces the number of data points that you feed into your loss function at each iteration of your algorithm.

Take a minute and think about why this might work. The reason that this still works well is that it's possible to extract samples from your training data set that, on average, balance each other out.

The pitfalls for sampling, and how to avoid them, is discussed in later modules. For now, keep in mind that your sampling strategy selects from your training set with uniform probability, so every instance in the training set has an equal chance of being seen by the model.

In ML, we refer to this practice of sampling from the training set during training as "mini-batching", and this variant of gradient descent as mini batch gradient descent. The samples themselves are referred to as "batches". Mini batch gradient descent has the added benefit, in addition to costing less time, of using less memory and of being easy to parallelize.

Now, a quick aside. You might hear people using the term "batch gradient descent". The batch there refers to batch processing, so batch gradient descent computes the gradient on the entire dataset! It's definitely not the same as mini-batch gradient

descent. Here, we're talking about mini-batch gradient descent.

Confusingly, mini-batch size is often just called batch-size. This is what TensorFlow calls it, and so this is what we'll call it too. In the rest of this specialization, when talking about batch size, it's talking about the size of the samples in mini-batch gradient descent.

So how big should those mini-batches be? Like the learning rate, batch size is another hyper-parameter and as such, its optimal value is problem-dependent and can be found using hyper-parameter tuning, which will be discussed in a later module. Typically, batch size is between 10 and 1000 examples.

Checking loss with reduced frequency

```
while loss is > Epsilon:
     derivative = computeDerivative()
     for i in range(self.params):
          self.params[i] = //
               self.params[i] //
                    - learning_rate //
                    * derivative[i]
     if readyToUpdateLoss():
          loss = updateLoss()
```

Popular implementations for readyToUpdateLoss():

- Time-based (e.g., every hour)
- Step-based (e.g., every 1000 steps)

The other knob you can turn to speed up model training is the frequency with which you check the loss. Recall that although it'd be great to simply check the loss on a subset of the data, this isn't a good idea. The implementation is quite simple: you introduce some logic such that our expensive computeLoss() function evaluates at reduced frequency. Some popular strategies for the readyToUpdateLoss() function are time-based and step-based. For example, once every 1000 steps. Or once every 30 minutes.

With the reduction of the frequency that you check the loss, and the introduction of mini-batching, you've now begun to decouple the two fundamental parts of model training, changing your model's parameters and checking to see when you've made the right changes.

Agenda

Defining ML models

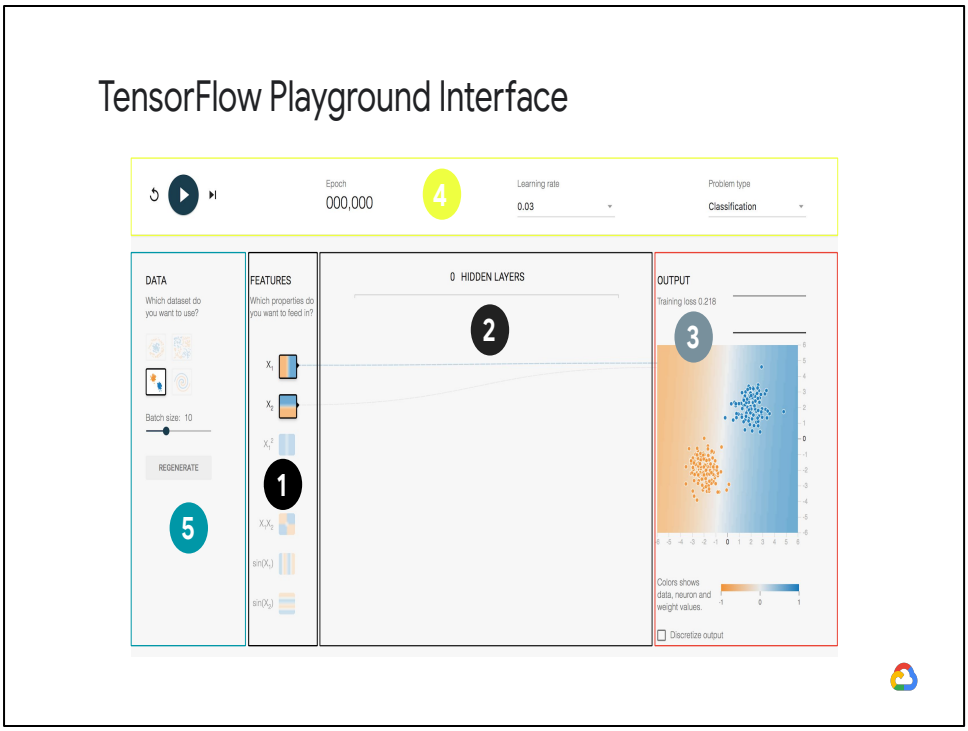
Introducing loss functions

Gradient descent

TensorFlow playground

Performance metrics

Now that you've learned how gradient descent works, you'll now see it in action using a tool that will allow you to see in real time many of the phenomena that have been discussed.



TensorFlow Playground is a powerful tool for visualizing how neural networks work. Now, you might be saying, “Hey, wait a minute! We haven’t introduced neural networks yet!” Don’t worry, that’s coming shortly. For reasons that will be explained, the simplest neural networks are mathematically equivalent to linear models. So, this tool is also well-suited to demonstrate what you’ve learned up until now.

You’re going to use it to experimentally verify the theoretical content introduced today so you can bolster your ML intuitions. You’ll see first-hand the impact of setting the learning rate and how ML models descend gradients. Connections to topics that will be explored in greater depth in this course and in later courses will also be called out.

First, let’s talk about the interface. Some of the features of the tool have been removed for now because they relate to material to be covering later, but there are still plenty of interesting knobs to turn.

First, there is the features column. These are the inputs your model sees. The coloring within each feature box represents the value of each feature: orange means negative and blue means positive.

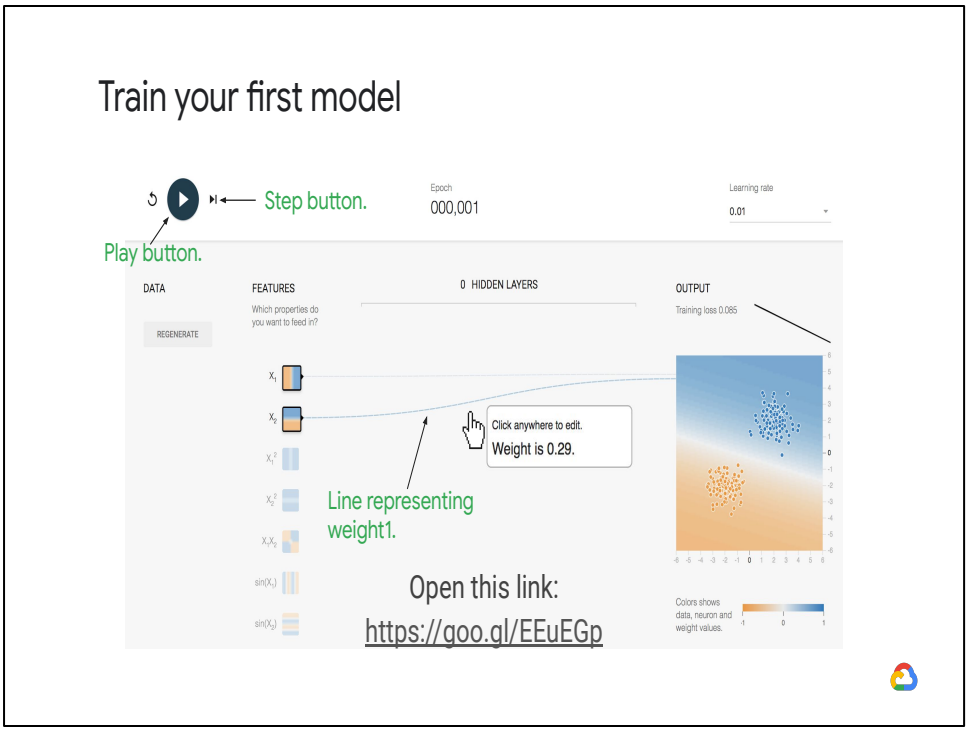
Secondly, there’s the Hidden Layers column, which you can think of as where the weights are. These are the model’s parameters; the bias terms are not depicted in this interface. If you hover over a weight line, you’ll see the value of the weight. As the model trains, the width and opacity of these lines will change to allow you to get a sense of their values quickly in aggregate.

Thirdly, there's the Output column, where you can see both the training data that the model has seen, as well as the model's current predictions for all of the points in feature space. You can also see the current training loss. As with the features, color is used to represent value.

Fourthly, the top control bar includes buttons for resetting training, starting training, taking a single step, the learning rate, and the ML problem type.

The Data column allows you to select different data sets and control the batch size.

Let's start by training a linear model to classify data.

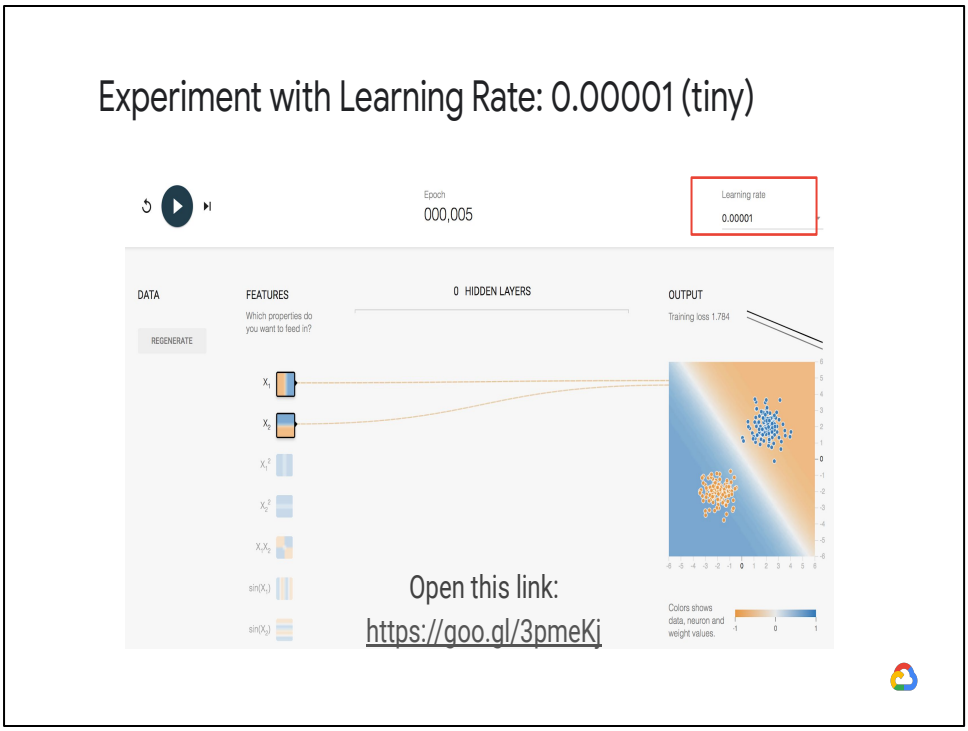


When you click the link, you'll be shown a TensorFlow Playground window with only the bare essentials (meaning that a few of the knobs have been hidden from you). Don't worry about the hidden layers for now.
In this configuration of the tool, the model accepts a feature vector, computes a dot product with a weight vector and adds a bias term, and uses the sign of the sum to construct the decision boundary. Consequently, you can think of this configuration as a linear model. [Note: this is only true because the activation function is set to be linear.]
You'll start with a model that will attempt to classify data that belong to two distinct clusters.

Click the step button, which is to the right of the play button, and note all the things that change in the interface: the number of epochs goes up by one, the lines representing weights change color and size, the current value of the loss changes, the loss graph shows a downward slope, and the output decision boundary changes.

Mouse over the line representing weight1 and note that you can see the value of this weight.

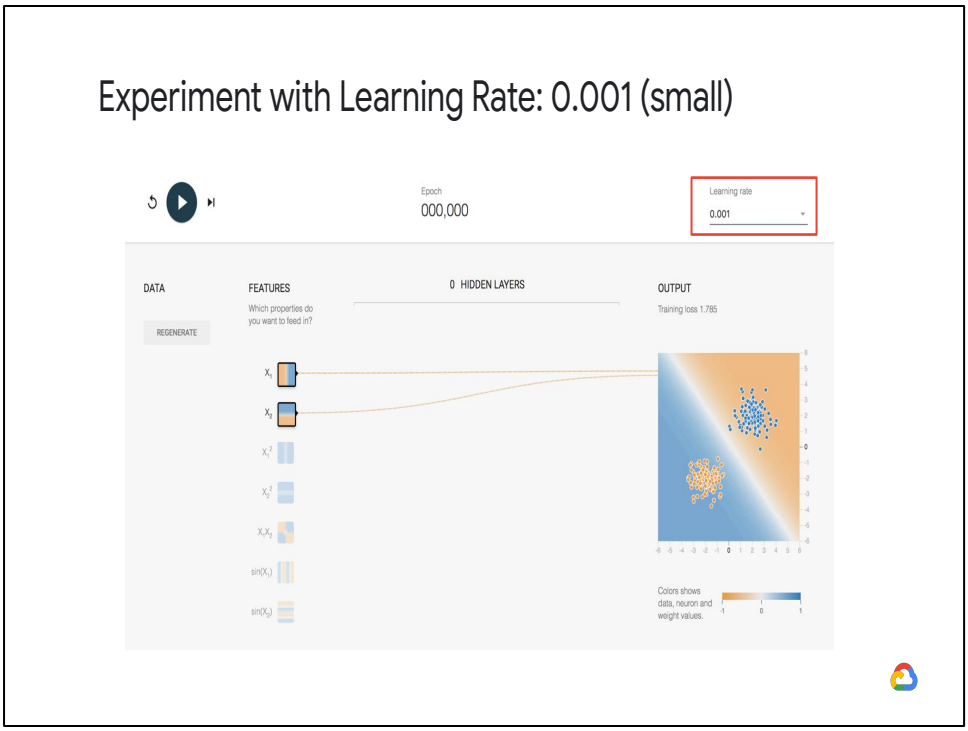
Click the play button to resume training, but pause soon after the loss drops below .002, which should occur before 200 epochs.
Congratulations, you've just trained your first model.



Let’s start adding some complexity. First, let’s see how three different learning rates affect the model during training.

Remember that learning rate is your hyperparameter, which is set before model training begins and which is multiplied by the derivative to determine how much you change the weights at every iteration of your loop.

Follow this link and start training the model with the very small learning rate that have been set as default for you. Wait until the loss reaches about 100 epochs, which should only be about two seconds, and then pause the model.

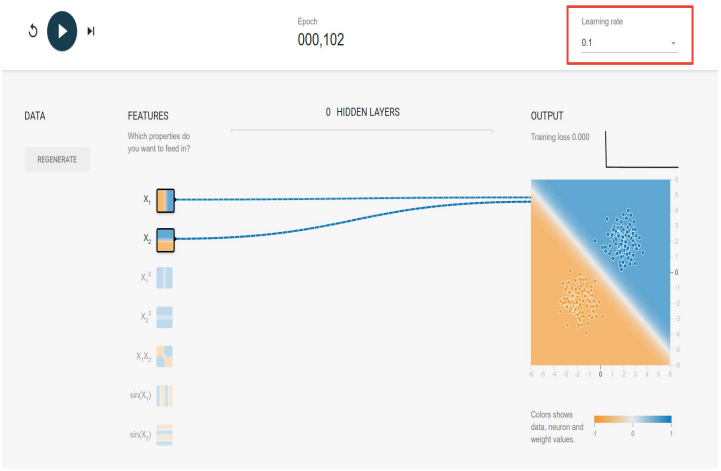


Now, increase the learning rate to .001 and restart training and once again stop around 100 epochs.

What is the loss? It should be substantially less. Note the value for weight1.

You might be tempted to conclude on the basis of this little experiment that .001 is a better learning rate. Universally speaking, it's not though, nor is it any worse. Instead, on the loss surface that results from this data set and this particular model, .001 proved to have been *faster to reach this level of loss*. Lower learning rates, though slower, are more likely to find minima. So, there's a tradeoff.

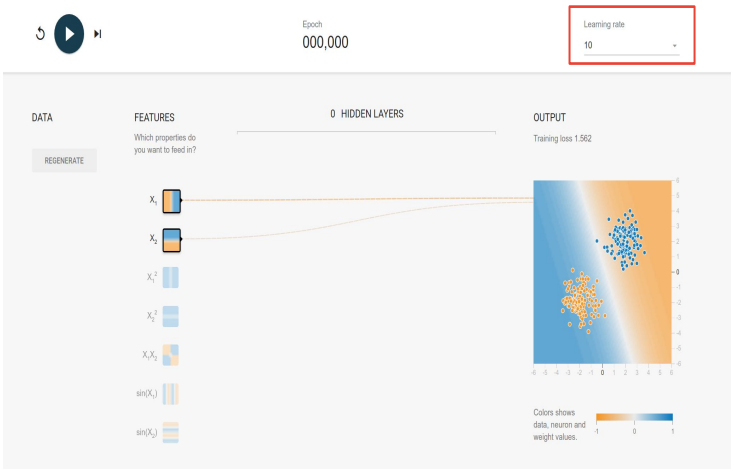
Experiment with Learning Rate: 0.1 (moderate)



Increase the learning rate to .1, restart the model training, and again train for 100 epochs.

How fast did the loss curve drop? It should have dropped very quickly.

Experiment with Learning Rate: 10 (huge)



Increase the learning rate to 10, restart the model training, and first take a single step using the step button. Note the magnitude of the weight. Now, continue training up until 100 epochs.

How fast did the loss curve drop? It should have dropped precipitously.

Let's put these observations together and see if we can explain them using what we've learning about optimization.

Experimenting with learning rate: Observations

Learning Rate	Weight1	Loss Over Time
.00001	.42	
.001	.65	
.1	1.0	
10	12.0	

This table shows an example of the results of the procedure you just used. Your results may look slightly different and that's okay. The reason that they may look different from the example is the same reason that they may look different if you rerun the experiment: TensorFlow Playground randomly initializes the weights. This means that your search starts off in a random position each time.

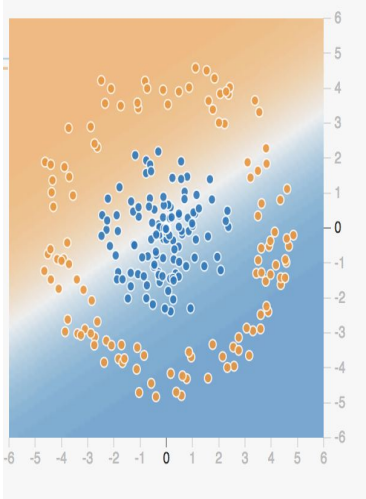
Let's talk about the Weight1 column. Note how the magnitude of the weights increased as the learning rate increase. Why do you think that is?

This is because the model is taking bigger steps. In fact, when the learning rate was 10, the first step changed the learning rate dramatically.

Let's talk about the rate of change. As the learning rate increased, the loss curve steepened. This is the same effect observed earlier, just through a different lens.

What about datasets that aren't easy to separate?

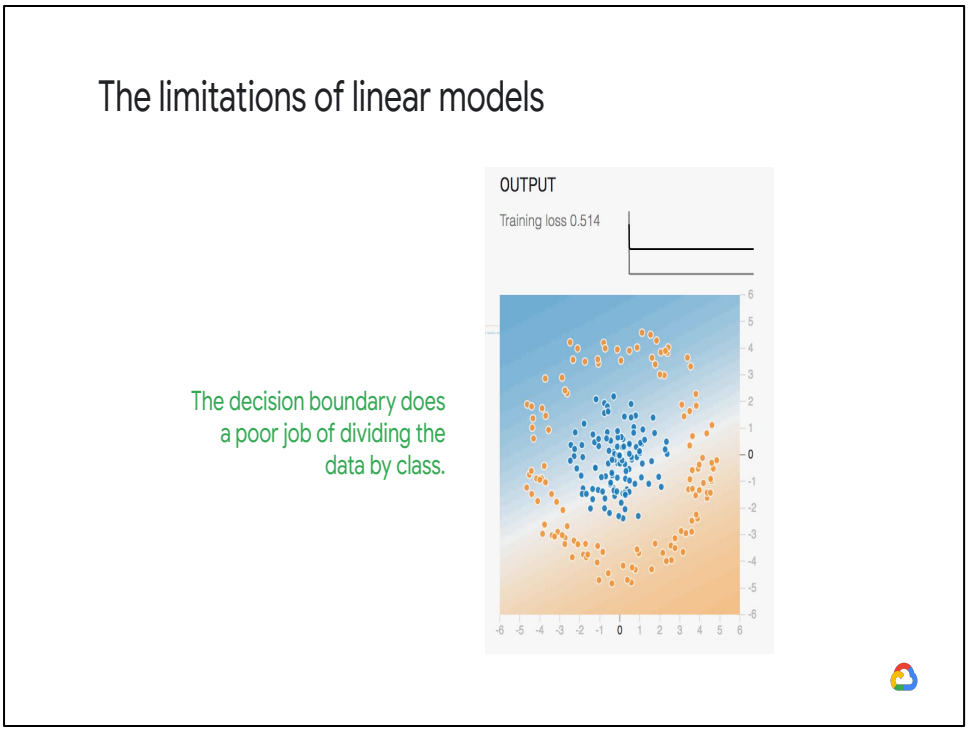
Open this link:
https://goo.gl/ou9iMB



Notice anything different about this dataset?

Click on the link and start training the model in the new window.

What do you observe about the loss and the graph of loss over time? Do you see any convergence toward zero over time?



If you've clicked the start training button directly, you should see output like what is shown here.

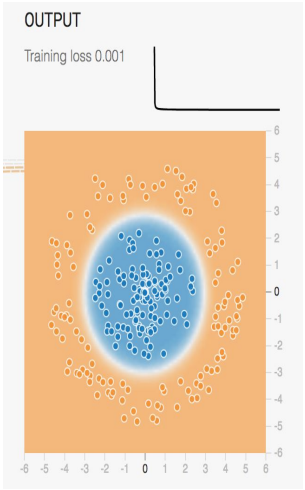
Note that the decision boundary does a poor job of dividing the data by class. Why might this be?

The reason is that the data have a non-linear relationship; that is, you can't draw a straight line dividing orange from blue. What this data calls for is a non-linear decision boundary, which in this case we intuitively recognize to be a circle around the blue data points.

However, all is not lost! By clicking on some of the boxes in the Input column, see if you can introduce new features that will dramatically improve performance.

Introducing non-linear features

Output after selecting the X1 squared and X2 squared features.



Hopefully, by now your output looks like this because you've selected the X1 squared and X2 squared features. Note how circular the decision boundary now is. How is it possible that a linear model can learn a non-linear decision boundary?

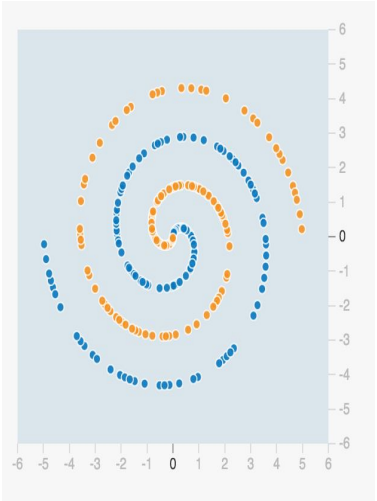
Recall that linear models learn a set of weights that they then multiply by their features to make predictions. When those features are first-degree terms like X and Y, the result is a first-degree polynomial, like 2X or ⅔ Y, and the model's predictions look like a line or a hyperplane.

But there is no rule that says that the features in a linear model must be first-degree terms! Just as you can take X squared and multiply it by 2, so too can you take a feature of any degree and learn a weight for it in a linear model.

Let's see how far we can take this idea.

Try learning a linear model for this dataset

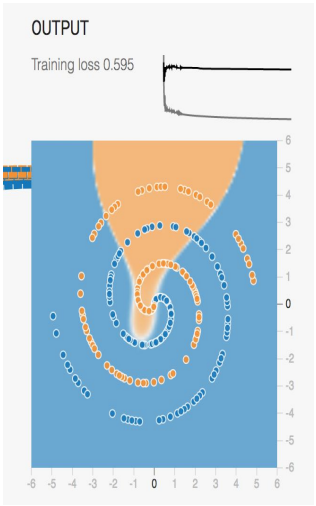
Open this link:
https://goo.gl/1v28Pd



Okay, now what about this one? The last time, we were able to find two non-linear features that made the problem linearly solvable. Will this strategy work here? Try it out!

Try learning a linear model for this dataset

Open this link:
https://goo.gl/1v28Pd



What you've now figured out is that using the feature options available to you and this type of model, this particular dataset is not linearly solvable; the model trained in this example has loss of about .6. However, this qualifier of feature options "available to you" is crucial because, in fact, there is a feature that would make learning this relationship trivial.

Imagine, for example, a feature that somehow unswirled the data so that blue and orange appeared simply as two parallel lines. These parallel lines would then be separable with a third line.

Finding features like that one are magical moments. They are also very difficult to anticipate, which is problematic.

However, even though you don't often find features that are as amazing as the ones seen in your toy examples, feature engineering, or the systematic improvement of or acquisition of new features, is an extremely important part of machine learning.

So, what can you do when your attempts to engineer new features for linear models fail? Use more complicated models.

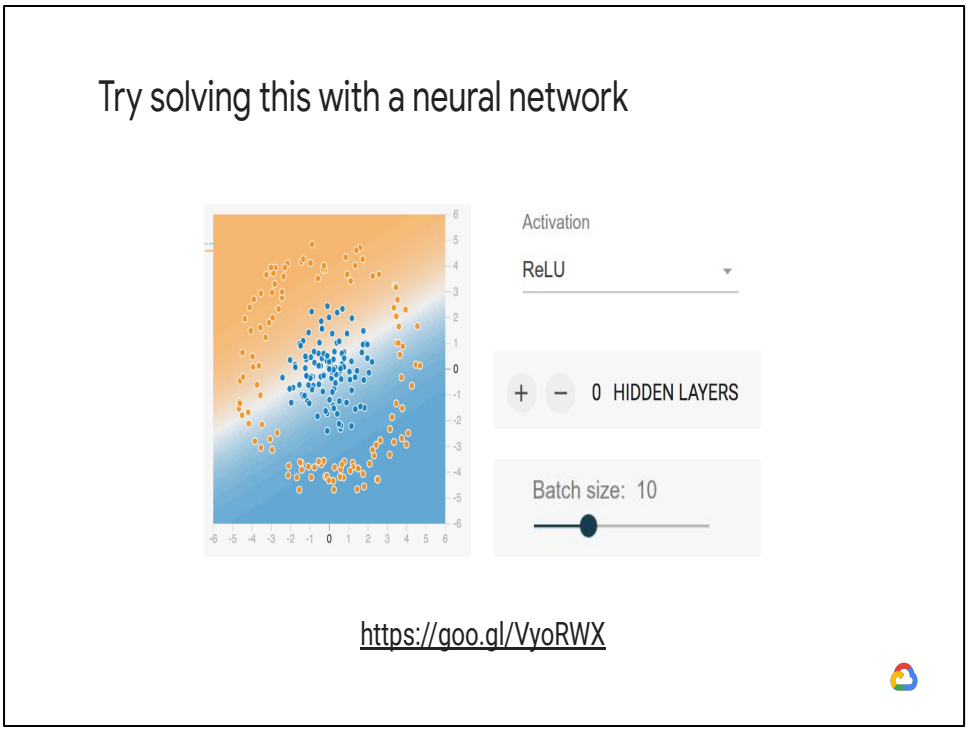
There are many types of models that are able to learn non-linear decision boundaries. In this course, we focus on neural networks, not because they are better--in fact, it's possible to prove that with respect to the set of all problems, no model type is better than any other--but because the sorts of business problems that are becoming

increasingly popular seem biased toward those where neural networks shine.

Lab

Develop an intuitive understanding of neural networks

Neural networks are becoming increasingly popular. You'll review how they work in detail later on. In this lab, the focus is on why neural networks are able to do the things that they can do.



You've already seen how a linear model can perform on this dataset. You will now see how a neural network does.

Before you do though, you need to review some additional features that have been enabled in TensorFlow Playground. The first feature enabled is Activation. Activation refers to the Activation Function. This is covered in more depth in the training *The Art and Science of ML*. The crucial point for now is that the choice of Activation Function is what separates linear models from neural networks. In the linear models you constructed previously, the activation function was, unbeknownst to you, set to be Linear.

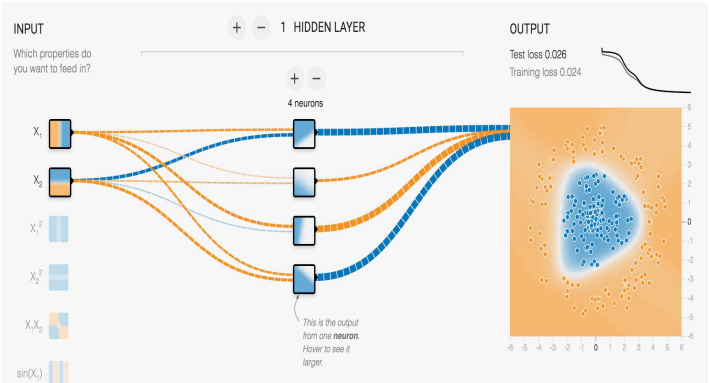
The second additional feature that has been enabled is the hidden layers feature. The hidden layers feature allows you to change the number of hidden layers and the number of neurons within each hidden layer. You can think of this as changing the number of transformations that the networks performs on your data. Each neuron in every hidden layer receives all of the output from the layer that precedes it, transforms that input, and passes output to all of the neurons in the subsequent layer. The shorthand way for describing the number of neurons and how they pass information to each other is the network's architecture.

Batch size has also been enabled, which you'll use in an experiment momentarily.

Follow the link on this slide and try to train a model that can classify this dataset. However, instead of introducing non-linear features, try to improve performance only

by changing the network's architecture. While we haven't actually explained how a neural network works, that's okay! For now, simply play around in the interface until you have a network that performs reasonably well.

More neurons means more input combinations (features)



https://goo.gl/SjQf5V

At this point, you should have a model that performs reasonably well, and the shape of the blue region in the output column should be a polygon. Let's take a look under the hood to get an intuition of how the model is able to do this.

Take a look again at the neurons in the first hidden layer. If you hover over each one, the output box changes to reflect what that neuron has learned. You can these neurons the same way you read the features and the output: the values of the features X1 and X2 are encoded in the position within the square and the color indicates the value that this neuron will output for that combination of X1 and X2.

As you hover over each of the squares in sequence, mentally, start imagining what they would look like superimposed on each other: blue atop blue becomes even bluer; blue atop white becomes light blue; blue atop orange becomes white.

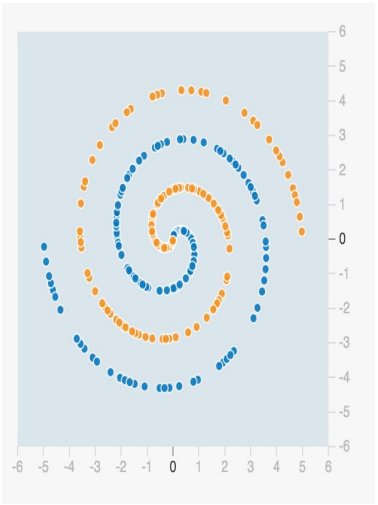
What you should start to see is how each neuron participates in the model's decision boundary, how the shape of the output is a function of the hidden layers. For example, this neuron contributes this edge to the decision boundary, while this neuron contributes this edge.

Given your knowledge of geometry, how small do you think you could make this network and still get reasonable performance out of it? To give you a hint, what's the simplest sort of shape you could draw around the blue dots that would still somewhat do the job? Experiment in TensorFlow Playground and see if your intuition is correct.

Background on Activation Functions:
https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

Will a set of lines work?

Open this link:
http://goo.gl/hrXd9T



So now, you've seen how output of the neurons in the first hidden layer of the network can be used together to compose a decision boundary.

What about those other layers? How does a neural network with one hidden layer differ from a neural network with many layers?

Click the link to start training a neural network to classify this spiral dataset.

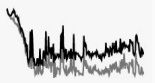
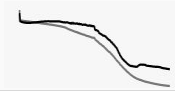
Let’s take this opportunity to understand more about how batch size affects gradient descent.

Set the batch size parameter to 1 and then experiment with neural network architectures until you find one that seems to work. Train your model for 300 epochs, and then pause and take note of the loss curve.

Now set the batch size parameter to 10 and then restart the training. Train your model for 300 epochs and pause. Once again, take note of the loss curve.

Finally, do this once more, but with the batch size equal to 30.

Experimenting with batch size: Observations

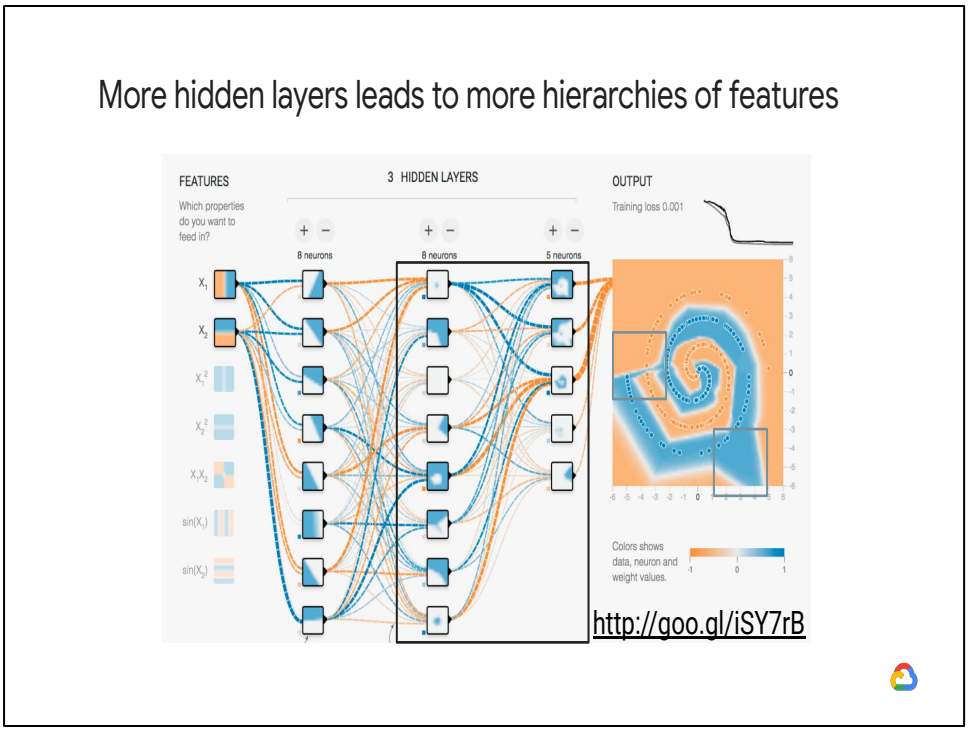
Batch Size	Loss Over Time
1	
10	
30	

What have you observed and how can you make sense of these observations given what you know?

What you should have seen is that there are marked differences in the smoothness of the loss curves: as batch size increased, so did the smoothness. Why might this be?

Think about how batch size effects gradient descent. When batch size is small, the model makes up an update to its parameters on the basis of the loss from a single example. Examples vary, however, and therein lies the problem. As batch size increases though, the noise of individual data points settles out and a clearer signal begins to take shape.

One thing you shouldn't conclude on the basis of these observations is that changes in batch size will have a simple effect on the rate of converge. As with learning rate, the optimal batch size is problem dependent and can be found using hyper-parameter tuning.



Now, your model should have finished training and it should look something like this.

The first thing to call out is the relationship between the first hidden layer and those that come after it. What should be apparent is that although the outputs from the neurons in the first hidden layer were basically lines, subsequent hidden layers had far more complicated outputs.

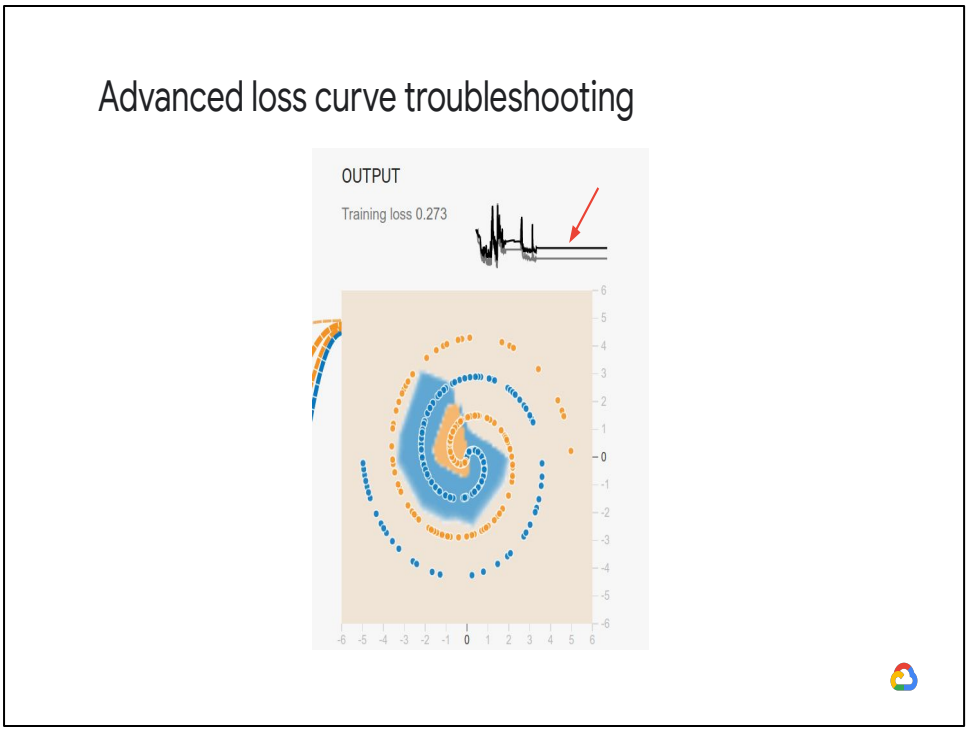
These subsequent layers build upon those that come before in much the same way that you did when stacking up the outputs of the hidden layer to understand the output. Consequently, you can think of a neural network as a hierarchy of features. And this idea, of taking inputs and transforming them in complex ways many times before ultimately classifying them, is typical of neural networks and represents a significant departure from the approach used classically in machine learning. Before neural networks, data scientists spent much more time doing feature engineering. Now, the model itself is taking some of that responsibility and you can think of the layers as being a form of feature engineering unto themselves. The next thing to call out are some strange things the model learned.

The model seems to have interpreted the absence of orange points in these two areas as evidence to support their blueness! We call mistakes like this, where the model has interpreted noise in the dataset as significant, “overfitting” and they can occur when the model has more decision making power than is strictly necessary for the problem. When models overfit, they generalize poorly, meaning that they don’t do well on new data, which are unlikely to have quite the same pattern of noise, even

though the underlying signal should remain.

How do we combat this? That will be covered in Generalization and Sampling.

Note: Even a complex non-linear pattern such as the double spiral could be classified by the simple neural network.



As you were experimenting with different neural network architectures, you may have trained models that entered terminal states like this one has. Note both the loss curve as well as the output. What did you do to fix them? And what's going on here?

While you may have changed your network architecture, often times you can fix problems like this simply by retraining your model. Remember, there are still parts of the model training process that are not controlled, such as the random seeds of your weight initializers.

The problem in this case is that you seem to have found a position on your loss surface that is small relative to its neighbors, but nevertheless much bigger than 0. In other words, you found a local minimum. Note how the loss over time graph actually reached a lower loss value earlier on in the search.

The existence and seductiveness of sub-optimal local minima are two examples of the shortcomings of this current approach. Others include problems like long training times and the existence of trivial but inappropriate minima.

These problems don't have a single cause, so your methods for dealing with them are diverse.

Advanced optimization techniques aim to improve training time and help models not to be seduced by local minima. Some of these will be reviewed later in the course.

Data weighting and oversampling and synthetic data creation aim to remove inappropriate minima from the search space altogether.

Performance metrics, which is covered in the next topic, tackle the problem at a higher level. Rather than changing the way you search or the search space itself, performance metrics change the way you think about the results of your search by aligning them more closely with what you care actually about. In so doing, they allow you to make better informed decisions about when to search again.

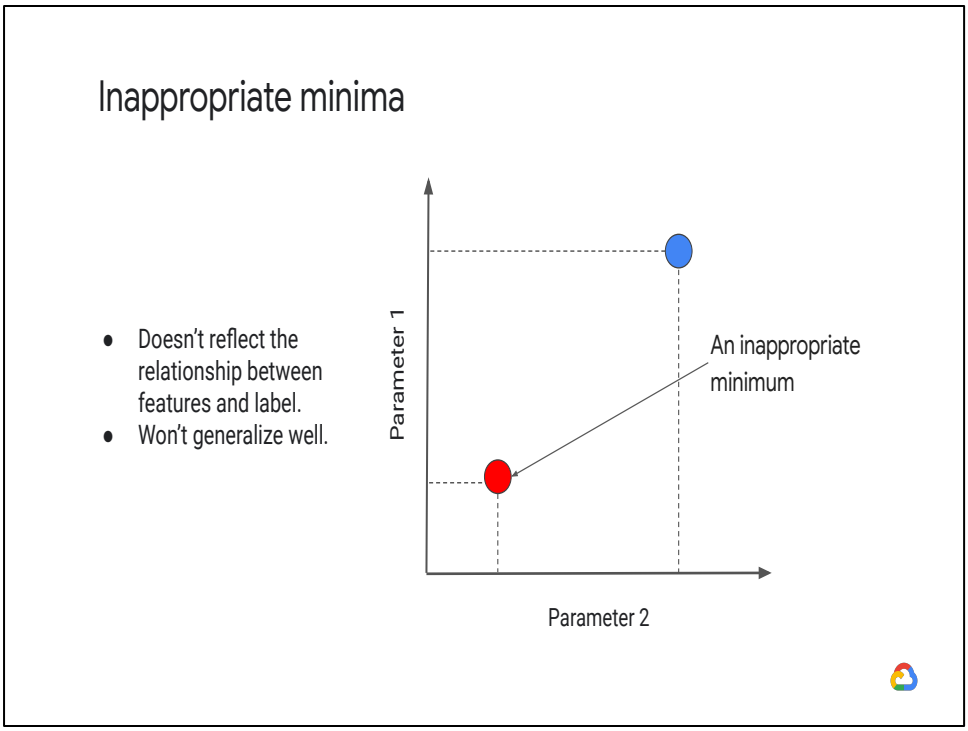
It would be great if you could create loss surfaces without such local minima in them. Unfortunately, this isn't possible. Minima like this arise in part because of the difference between what you must optimize for, your loss function, and what you actually care about, namely making a decision in the real world.

There are two strategies for mitigating this problem. The first is to modify the gradient descent algorithm in order to better avoid local minima. Such advanced optimization techniques are reviewed in a later course.

The other strategy is to consider the real-world context in which a model operates in a more formal way. This is what you'll do in the next topic on Performance Metrics.

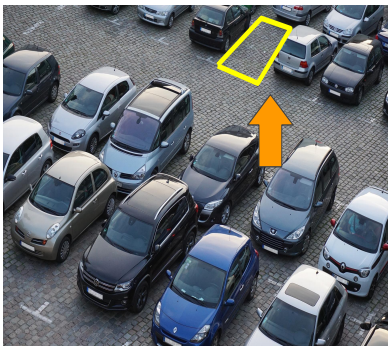
In the previous topic, you trained models in your browsers using gradient descent and the models that you created were able to learn complex, non-linear relationships using a learned hierarchy of features. However, you discovered at the end of the topic that your current approach suffers from problems, the consequences of which include things like long training times, suboptimal minima, and inappropriate minima.

This topic reviews what exactly an inappropriate minimum is, why they exist, and how performance metrics help you get better results.



So, what is an inappropriate minimum? You can think of them as points in parameter-space that reflect strategies that either won't generalize well, or doesn't reflect the true relationship being modeled, or both.

Skewed data can make inappropriate strategies seductive



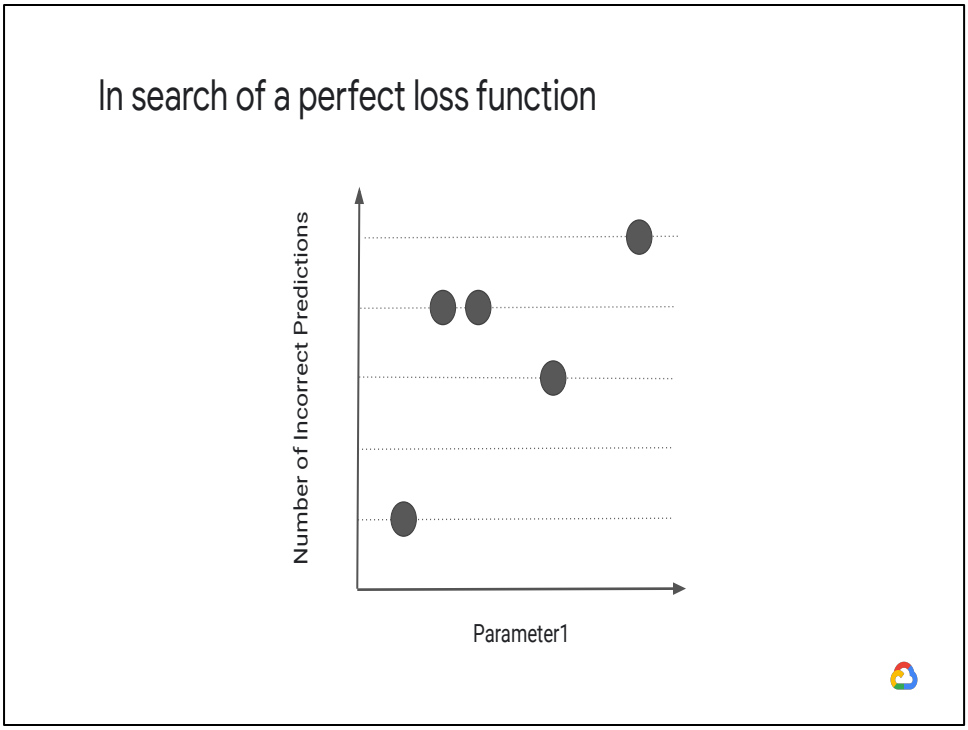
1000 parking spaces.
990 of them are **taken**.
10 are **available**.

An ML model that always reported that a space was occupied would be right 99/100 times.

Say you're training a model to predict whether a parking spot is vacant from an image of a parking lot. One inappropriate strategy would be to simply predict that every space is occupied. With a dataset composed of an equal number of positive and negative examples, such a strategy would never survive the optimization process. However, when datasets are skewed, and contain far more of one class than another, then suddenly strategies like these can become much more seductive.

Such a strategy doesn't make an effort to understand the true relationship between the features and the label, which you expect would have something to do with the visual characteristics of an empty space. Consequently, it won't generalize well to new parking lots where the underlying relationship will be the same, but the proportion of vacant spots may not be.

https://pixabay.com/en/parking-autos-vehicles-traffic-825371/ (cc0)



It's tempting to think of the existence of inappropriate minima as a problem with your loss function. If only you had the perfect loss function, one that rewarded the truly best strategies and penalized the bad ones, then wouldn't life be grand!

Sadly, this just isn't possible. There will always be a gap between the metrics we care about and the metrics that actually work well with gradient descent.

For example, assume you're still classifying parking spaces. A seemingly perfect loss function would minimize the number of incorrect predictions. However, such a loss function would be piecewise; that is, the range of values it could take would be integers, not real numbers. And surprisingly, this is problematic.

The issue boils down to differentiability: gradient descent makes incremental changes to your weights. This in turn requires that you can differentiate the weights with respect to the loss. Piecewise functions, however, have gaps in their ranges, and while TensorFlow can differentiate them, the resulting loss surface will have discontinuities that will make it much more challenging to traverse.

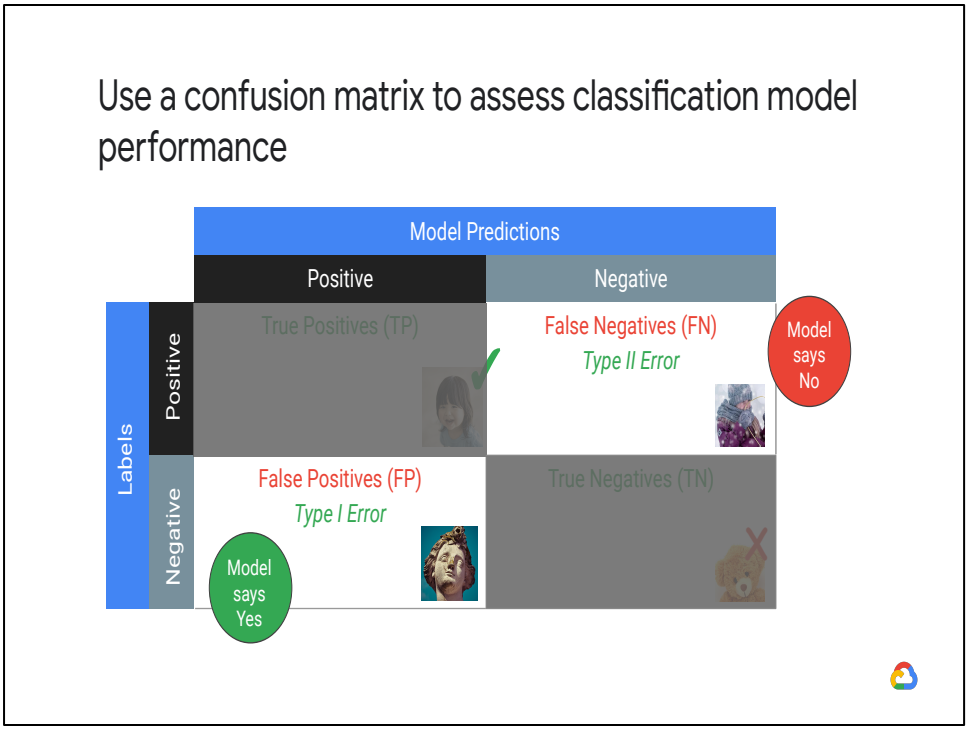
Performance metrics allow you to measure what matters

Loss Functions	Performance Metrics
During training.	After training.
Harder to understand.	Easier to understand.
Indirectly connected to business goals.	Directly connected to business goals.

You need to reframe the problem. Instead of searching for the perfect loss function during training, you're instead going to use a new sort of metric after training is complete that will allow you to reject models that have settled into inappropriate minima.

Such metrics are called performance metrics. Performance metrics have two benefits over loss functions. Firstly, they are easier to understand. This is because they're often simple combinations of countable statistics. Secondly, performance metrics are directly connected to business goals. This is a subtler point, but it boils down to the fact that while loss and the business goal that is being sought will often agree, they won't always agree; sometimes, it'll be possible to lower loss while making little progress toward, or perhaps even straying farther from, a business goal.

Three performance metrics will be reviewed, together with when you'll want to use them: Confusion Matrices, Precision, and Recall.



This is a matrix you might have seen before when Inclusive ML and facial recognition was discussed in earlier training. In that example, a face-detection ML model was looked at which incorrectly predicted a statue as a human face (which is called a False Positive), and also missed an actual face in the dataset when it was obscured with winter clothing (this miss is called a False Negative).

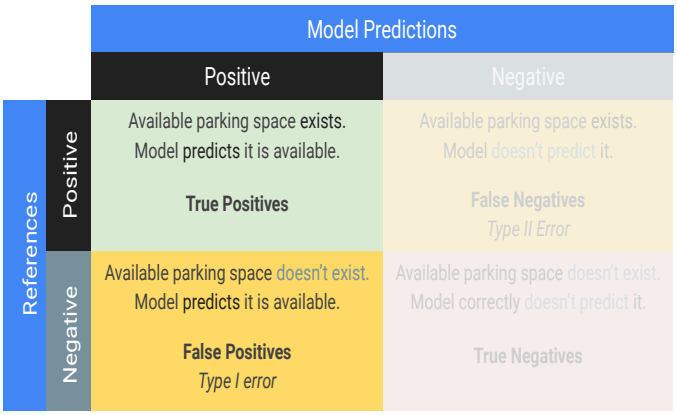
A confusion matrix like this one will allow you to quantifiably assess the performance of your classification model. But now you have 4 numbers (one for each quadrant), and business decision makers want to see only one. Which one do you present?

To explore this a bit more, take a look at another photo classification example.

Notes:
Get the outcome for each image in that dataset, then add them up
Confusion matrix is great, but it's 4 numbers and business decision makers want to see only one. Which one?

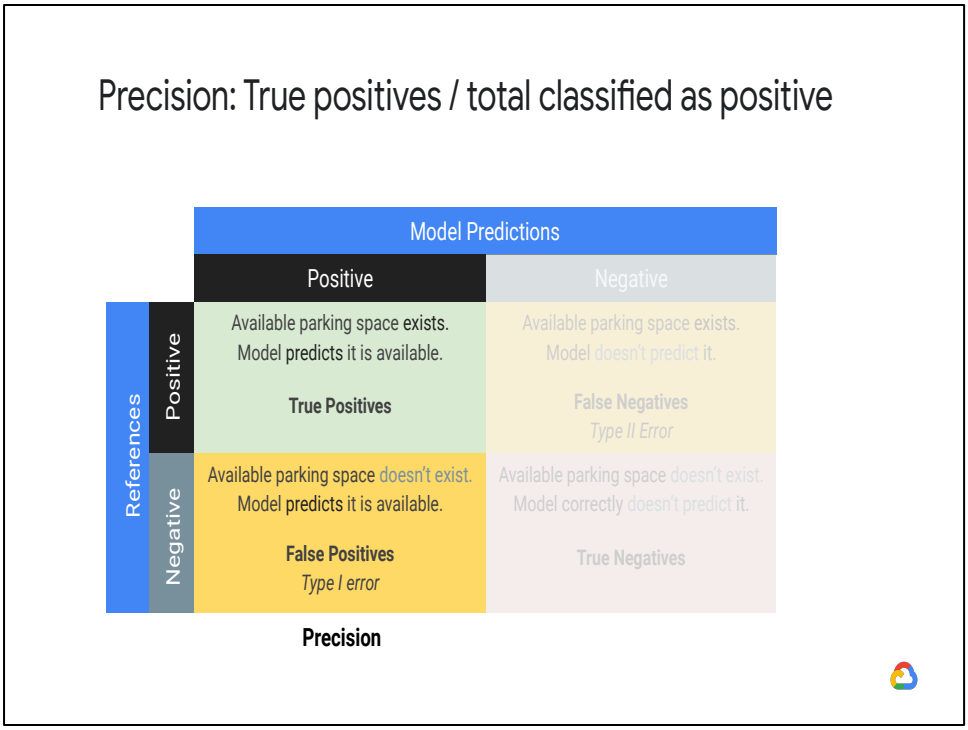
Image: clipart https://pixabay.com/en/cartoon-cat-cute-1292872/

True and false positives when predicting parking spots



If you know that a parking spot is available (reference is positive), and the model also predicts that it's available, this is called a true positive.

If you know that a parking space is not available, but the model predicts that it is, this is called a False Positive or Type 1 error.

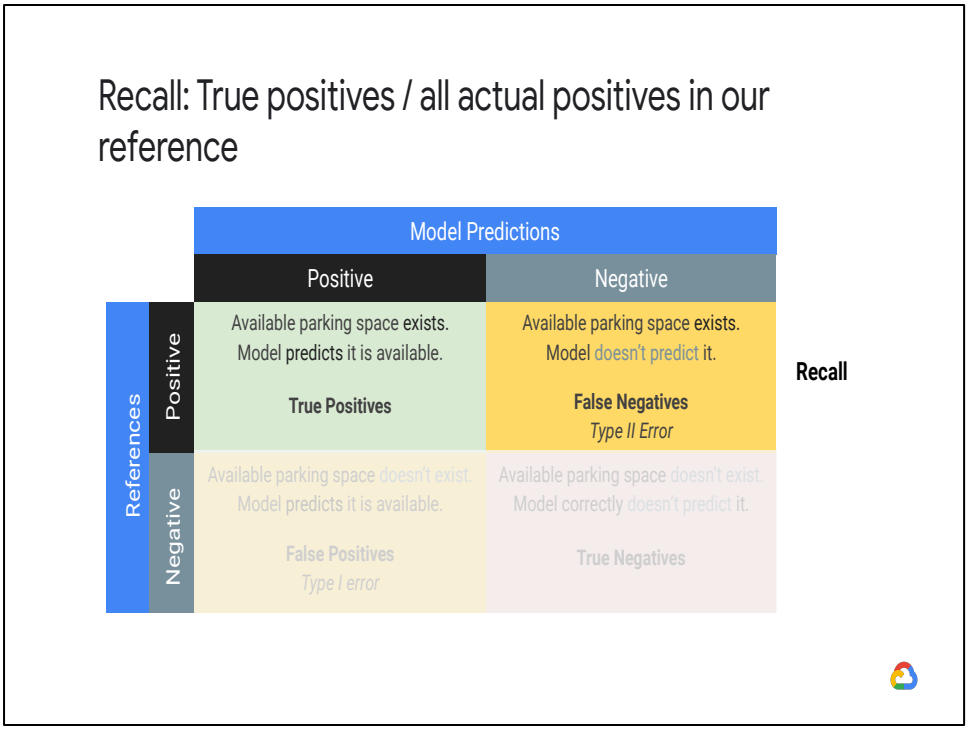


To compare how well your model did with its positive predictions, a metric called precision is used.

With high precision, if you say a parking space is available you're really sure it is. A precision of 1.0 means that of the available spaces you've identified, all of them are actually available (but you could have missed other available spaces, which are called False Negatives).

It's formally defined as the number of true positives divided by the total number classified as positive.

Looking at the matrix, an increase in what factor would drive down precision? An increase in False Positives. In your parking lot example, the more the model predicts spaces as available which really aren't, the lower your precision.



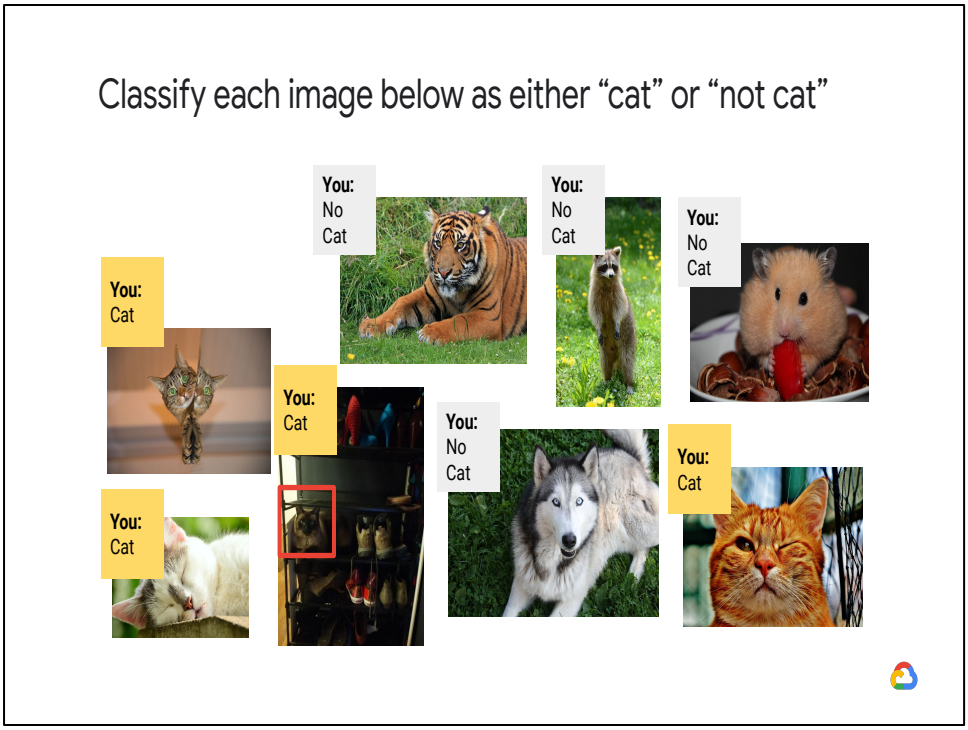
Recall is often inversely aligned with precision. With high recall, you're rewarded for finding lots of actually available spots. A recall of 1.0 would mean you found all 10 out of 10 available parking spots (but you also could have had many spots you thought were available but weren't -- these are called false positives).

What was the recall of your parking lot example? Remember you 10 actually had available spaces and your model identified only 1 as available.

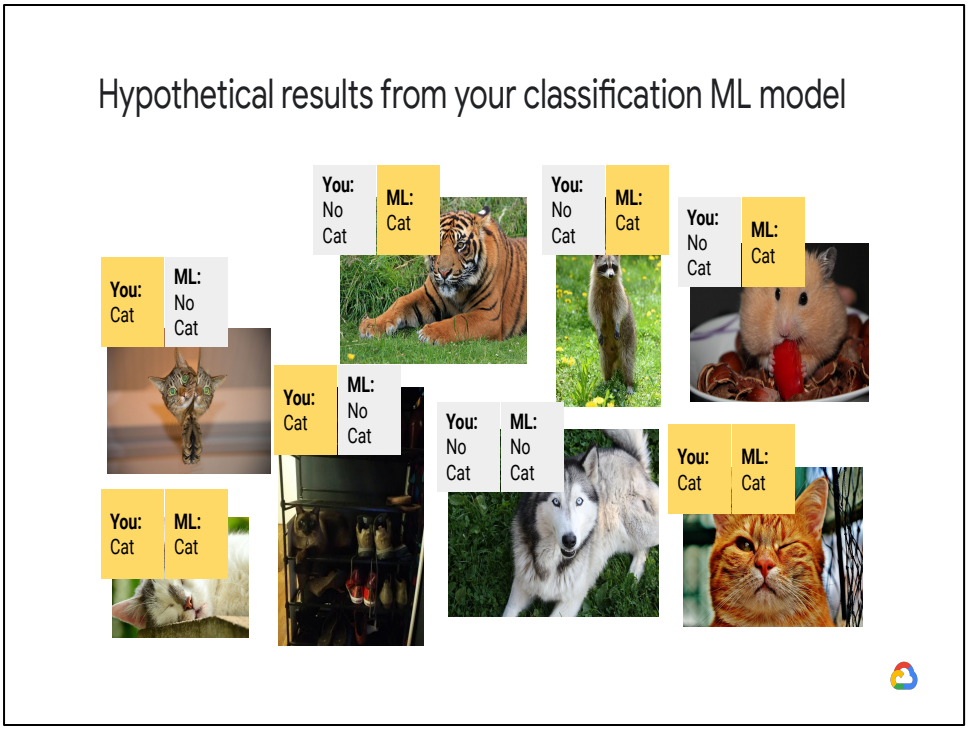
The answer: Recall = 1/10 = 0.1.

Here you’re presented with a dataset of domestic house cats and “not cats”. Take a quick minute and spot the images that are cats or not cats, as this will form your ground truth or labeled dataset.

Image: cat with shoes courtesy of course author, Cassie Kozyrkov
Images: https://pixabay.com/en/cat-cat-face-sleep-exhausted-1551810/
https://pixabay.com/en/cat-pet-mirror-697113/
https://pixabay.com/en/tiger-sumatran-sumatran-tiger-164905/
https://pixabay.com/en/cat-wink-funny-fur-animal-red-1333926/
https://pixabay.com/en/huskies-dog-animal-blue-eyes-view-1370230/
https://pixabay.com/en/goldhamster-hamster-animal-nuts-943373/
https://pixabay.com/en/racoon-animal-garden-summer-1453600/



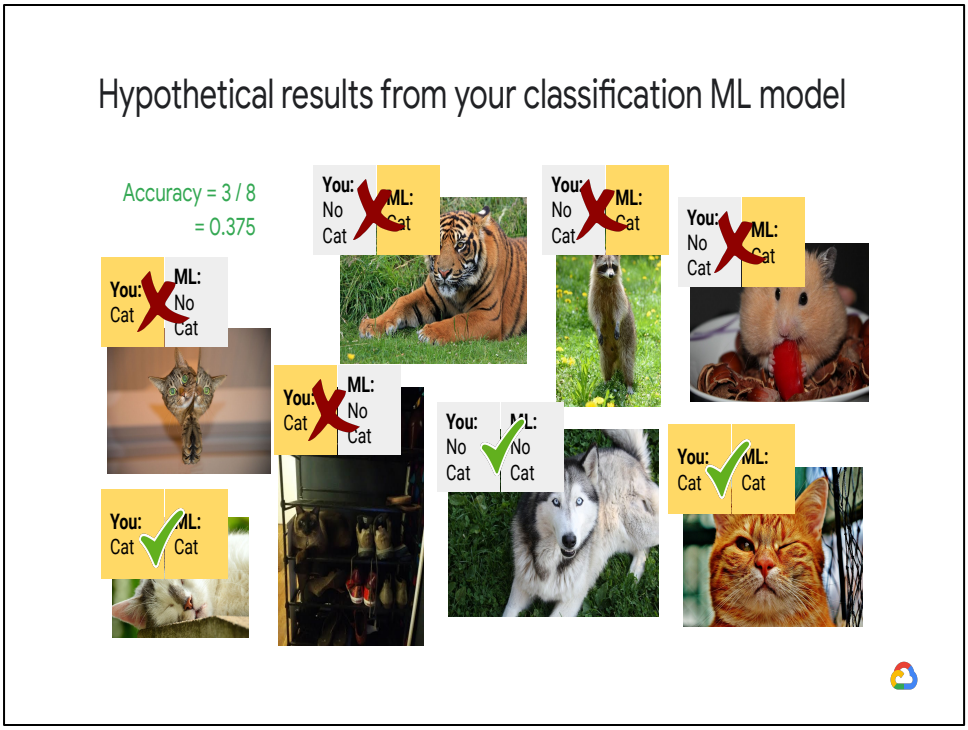
Hopefully, you found all the domestic cats as correctly shown here. Note the hidden cat in red, and that the tiger, for our purposes, is not classified as a domestic house cat.



And here's what your model came up with. Compare the results against what you know to be true.

Now you have your properly labeled data points side-by-side with your model predictions.

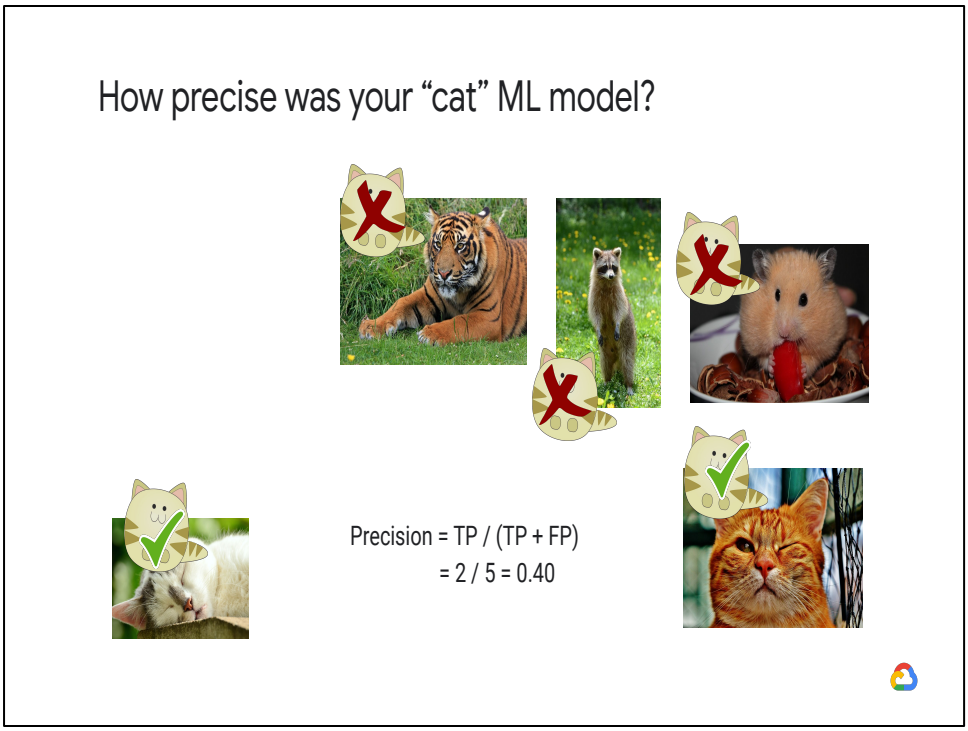
In total, you have 8 examples, or instances, that you showed the model. How many times did the model predict 'cat' correctly or 'not cat' correctly?



How many times did the model make incorrect predictions?

Three out of a total of 8 were accurately predicted. This gives your model an accuracy of .375

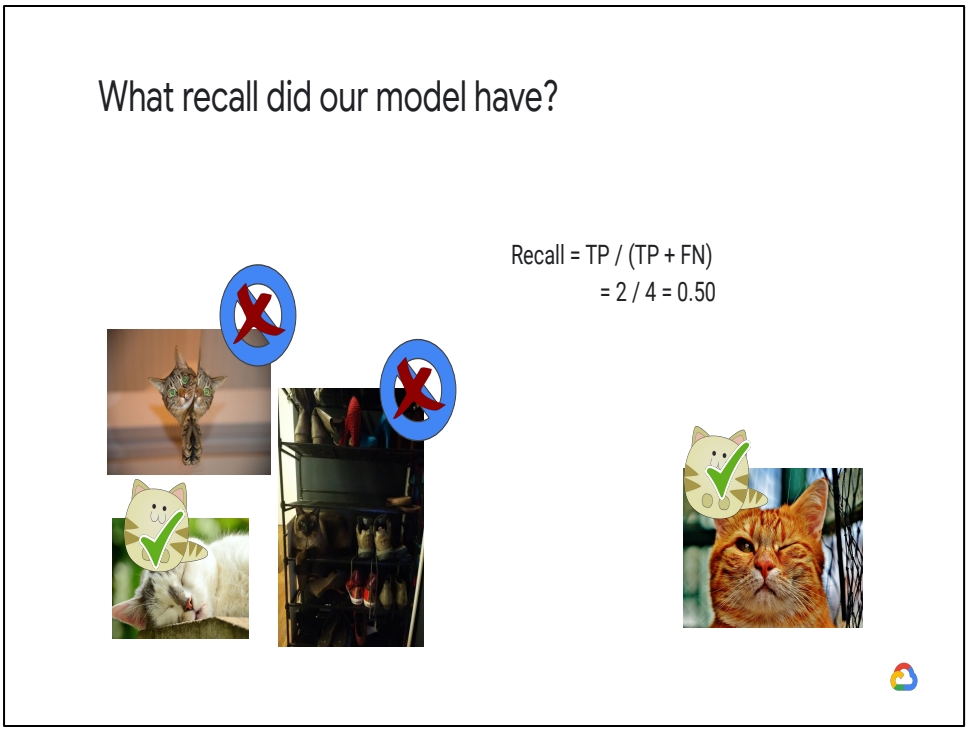
Is this accuracy metric the only or best way to describe your model performance? Before exploring other ways, there's a common pitfall to discuss.



Revisiting your “cat” and “no cat” example, with what precision did the model identify cats out of all those classified as positive by either your intuition or the model?

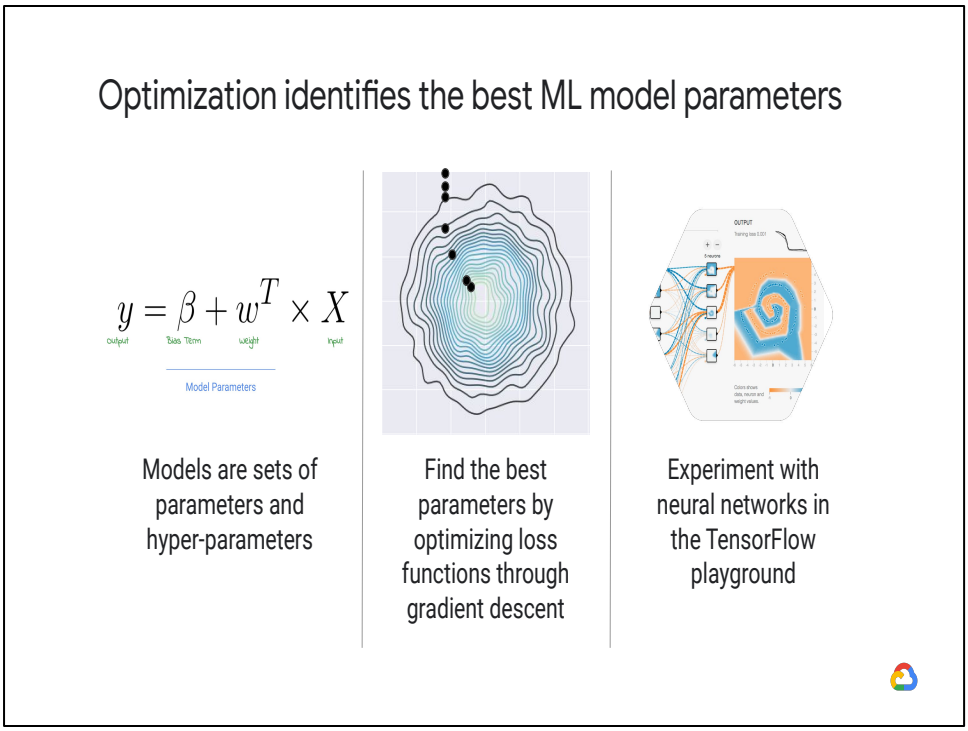
The five images here were in the positive class. How many are actually domestic cats?

Two out of the 5, or a precision rate of .40.



Recall is like a person who never wants to be left out of a positive decision. Here you see all the true labeled examples of cats, and the model's performance against them. What was the recall? Or said another way, how many actual true positives did the model get right?

The model only got two of the four actual cats correct, or a recall of .50.



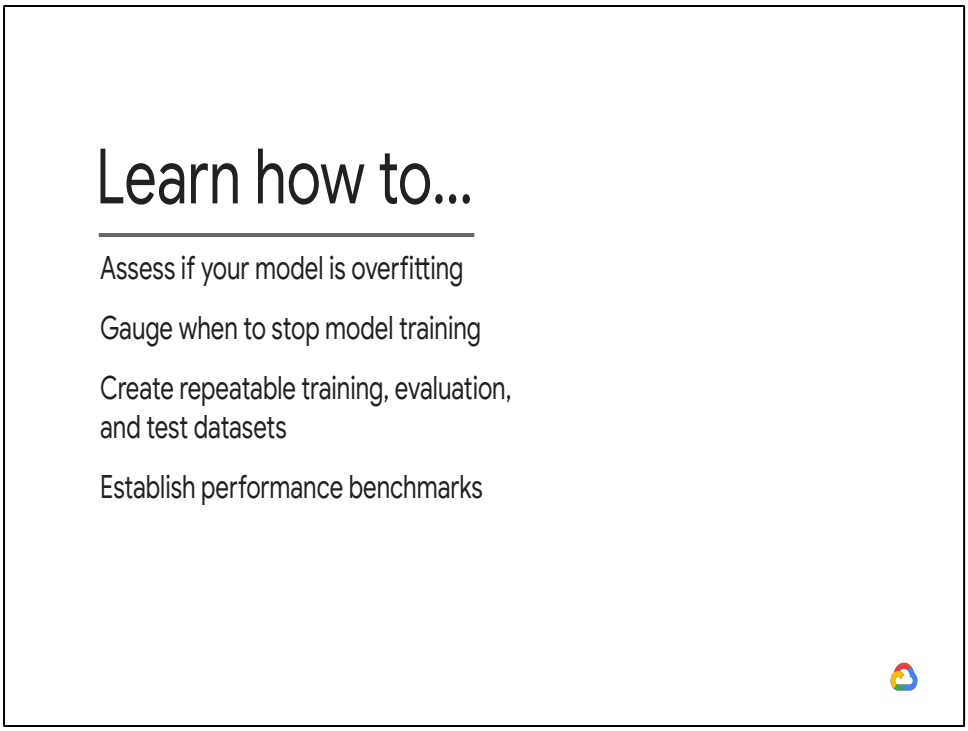
This is a quick wrap-up of what you've learned so far about optimization.
First, ML models were defined as sets of parameters and hyper-parameters, which started your search in the parameter space for the parameters that best modelled reality as seen in your training dataset.
Next, you were introduced to Loss Functions, which is how you quantifiably measure and evaluate the performance of your model with each training step. Two examples of specific Loss Functions discussed were RMSE for your baby weight linear regression model, and cross-entropy for your "face" or "not face" classification model.

You learned how to traverse your Loss Surfaces efficiently by analyzing the slopes of your Loss Functions, which provide you direction and step magnitude. This process is called Gradient Descent.

You experimented with different ML models inside of the TensorFlow Playground and saw how the linear models can learn non-linear relationships when given non-linear features, and how neural networks learned hierarchies of features. You also saw how hyper-parameters, like learning rate and batch size, affect gradient descent.
You then walked through how to choose between accuracy, precision, and recall for classification model performance depending on which problem you are trying to solve.
As you saw throughout this module, your labeled training dataset was the driving force that the model learned from. In the next module, you will cover how to effectively split your full dataset into training and evaluation and the pitfalls to avoid along the way.

Generalization and Sampling

Welcome to ‘Generalization and Sampling.’



So far in this course we have discussed ML model training and experimented with model training inside the TensorFlow playground.

Now it's time to answer a rather weird question: when is the most accurate ML model **not** the right one to pick? As we hinted at in the last module on Optimization -- simply because a model has a loss metric of 0 for your training dataset does not mean it will perform well on new data in the real world.

You've got to realize that the best ML model is not necessarily the one that performs the best on your training dataset but the one that performs best on unseen data. Your main concern should be how your model performs in production. This implies data that your model has yet to see. So how would you measure model performance on unknown data? Well, you need some data that is not shown to the model during training. After you train the model, you can evaluate on this "held-out" data.

You will learn how to assess whether your model is overfitting and how to gauge when to stop model training.

The second part of this module is about how you create this unknown dataset in the first place. Naturally, you don't have unknown data but what you do have is a training dataset that you can then split into separate training and evaluation datasets. You can experiment and train your model with one dataset and then, when you're ready to measure how the model will perform in the real world, you can use your other dataset.

You will learn how to create repeatable training, evaluation, and test datasets and establish performance benchmarks.

Agenda

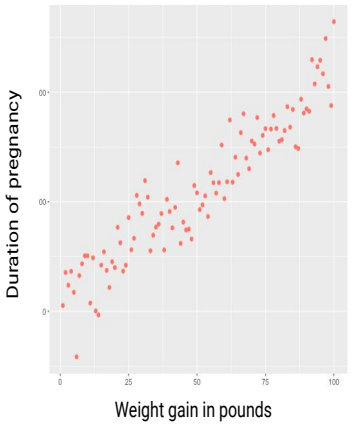
Generalization

Sampling

Let’s first address generalization, which will help us answer the question about the most accurate ML models not always being the right choice.

Suppose we want to predict duration of pregnancy based on mother's weight gain in pounds

What is the error measure to optimize?

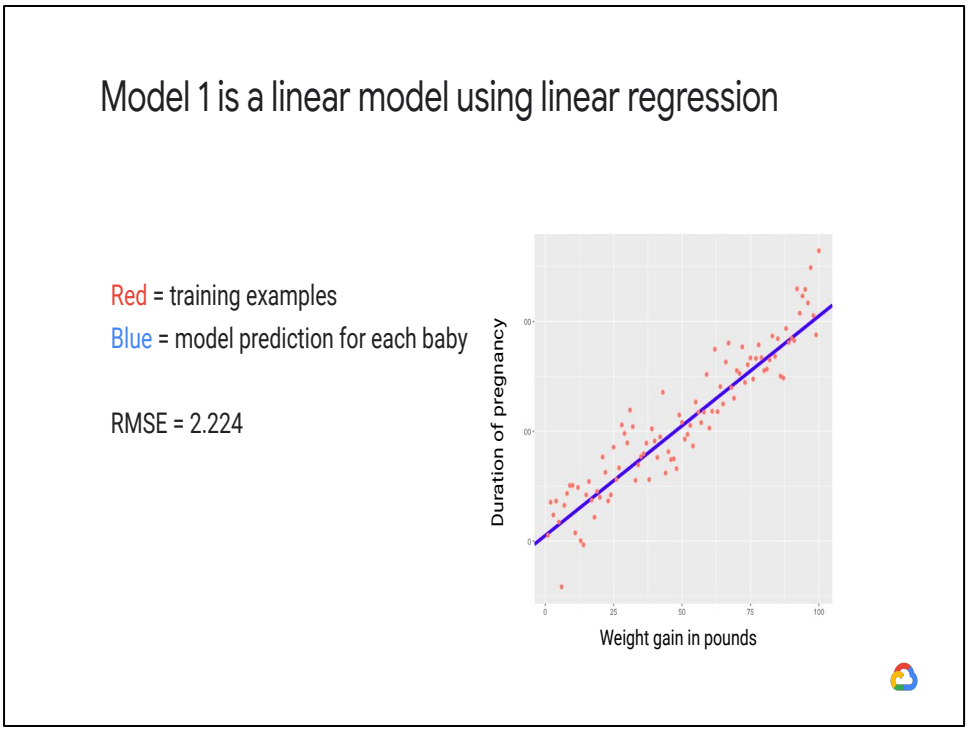


Here we find ourselves once again our familiar natality dataset, but this time we are using the mother's weight gain on the X-axis to predict the duration of the pregnancy (on the Y-axis).

What do you observe about the pattern you see in the data? It looks very strongly correlated (the more weight gained, the longer the pregnancy duration, which intuitively makes sense as the baby grows).

To model this behavior and prove a correlation, what model do you typically want to call on first? A simple linear regression model

And as we covered, for regression problems, the loss metric you want to optimize is typically Mean Squared Error (MSE) or Root Mean Squared Error (RMSE).



The Mean Squared Error tells you how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the “errors”) and squaring them. The squaring is necessary to remove any negative signs. MSE also gives more weight to larger differences.

Taking the square root of the MSE gives us the RMSE, which is simply the distance, on average, of a data point from the fitted line, measured along a vertical line. The RMSE is directly interpretable in terms of measurement units, and so is a better measure of goodness of fit than a correlation coefficient.

For both error measures a lower value indicates a better performing model. The closer the error is to zero, the better.

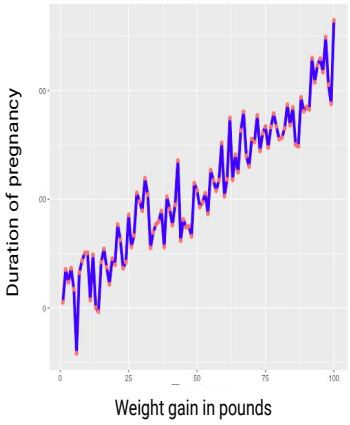
Here we’re using a Linear Regression model, which simply draws the line of “best fit” to minimize the error. Our final RMSE is 2.224. Let’s say that, for our problem, this is pretty good.

Model 2 has more free parameters

RMSE = 0

Which model is better?

How can you tell?



But look at this! What if we use a more complex model? A more complex model has more free parameters. In this case, these free parameters let us capture every squiggle of the dataset. Doing so ...

We reduced our RMSE all the way down to 0 -- the model is now perfectly accurate. Are we done? Is this the best model?

People intuitively feel that there is something fishy about Model 2.

But how can we tell? In ML, we often have lots of data, and no such intuition. Is an NN with 8 nodes better than an NN with 12 nodes? The 12-nodes version has lower RMSE … so should we pick it? But what if we try 16 nodes, and it has even lower RMSE?

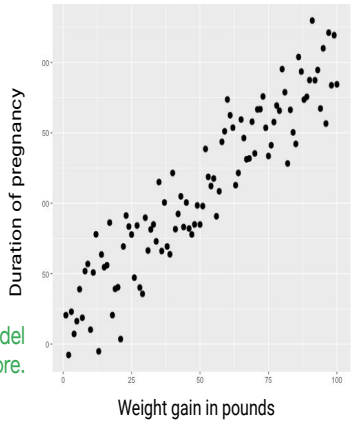
The example here might be a polynomial of 100th order or a neural network with lots of nodes. As you saw in the spiral example at the end of the last lecture on Optimization, a more complex model has more parameters that can be optimized. While this can help it fit more complex data, it might also help it memorize simpler datasets.

At what point do we stop and say a model is now simply memorizing the data and overfitting?

Does the model generalize to new data?

Need data that were not used in training.

New data the model hasn't seen before.



One of the best ways to assess the quality of a model is to see how well it performs against a new set of data that it has not seen before. Then we can determine whether the model "generalizes" well across new data points.

Let's check back on the linear regression model and neural network models and see how they are doing...

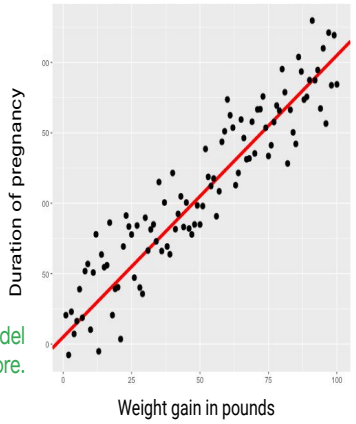
Model 1 generalizes well

Old RMSE = 2.224

New RMSE = 2.198

Pretty similar = good

New data the model hasn't seen before.



Our linear regression model on the new data points is generalizing well. The new Root Mean Squared Error is comparable to what we saw before (no surprises is a good thing; we want consistent performance out of our models).

Model 2 does not generalize well

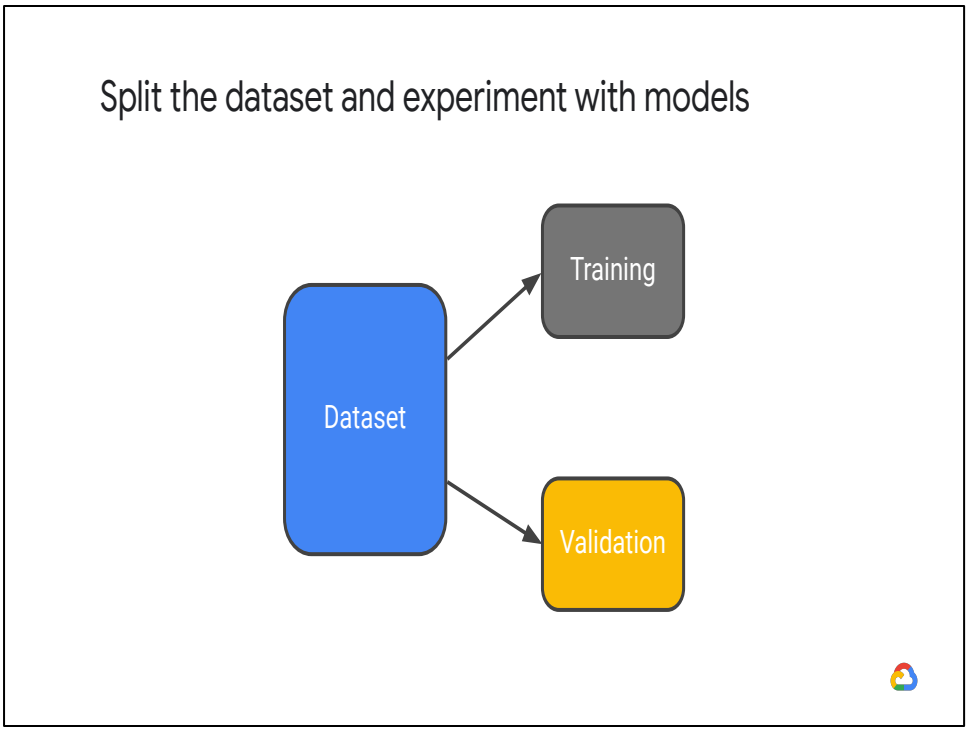
Old RMSE = 0

New RMSE = 3.2

This is a red flag

New data the model hasn't seen before.

Looking back at model 2, we can see that it does not generalize well at all on the new training dataset. The RMSE jumped from 0 (perfect accuracy) to 3.2, which is a big problem. The model was completely overfitting on the training dataset it was provided, and that proved to be too brittle (not generalizable) when new data was provided.

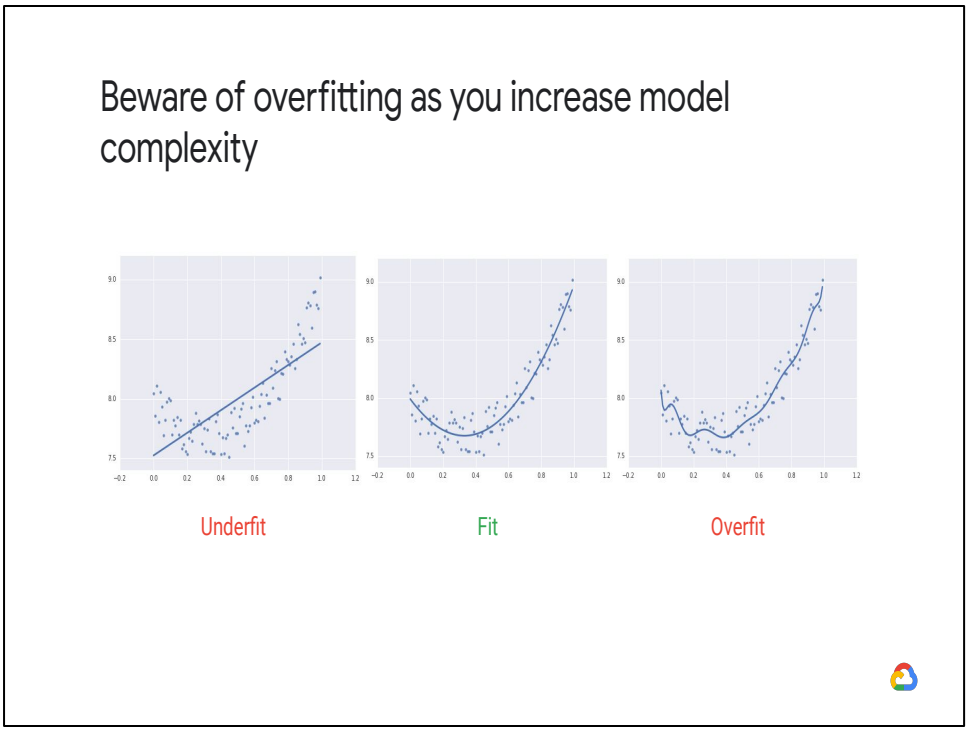


Now you may be asking, how can I be sure that my model is not overfitting? How do I know when to stop training?

The answer is simple: Split your data!

By dividing your original dataset into completely separate and isolated groups, you can iteratively train your model on your training dataset and then compare its performance against an independent validation dataset.

Models that generalize well have similar error values across training and validation. As soon as you start seeing your models not perform well against your validation data (for example, if your loss metrics start to increase), it's time to stop.



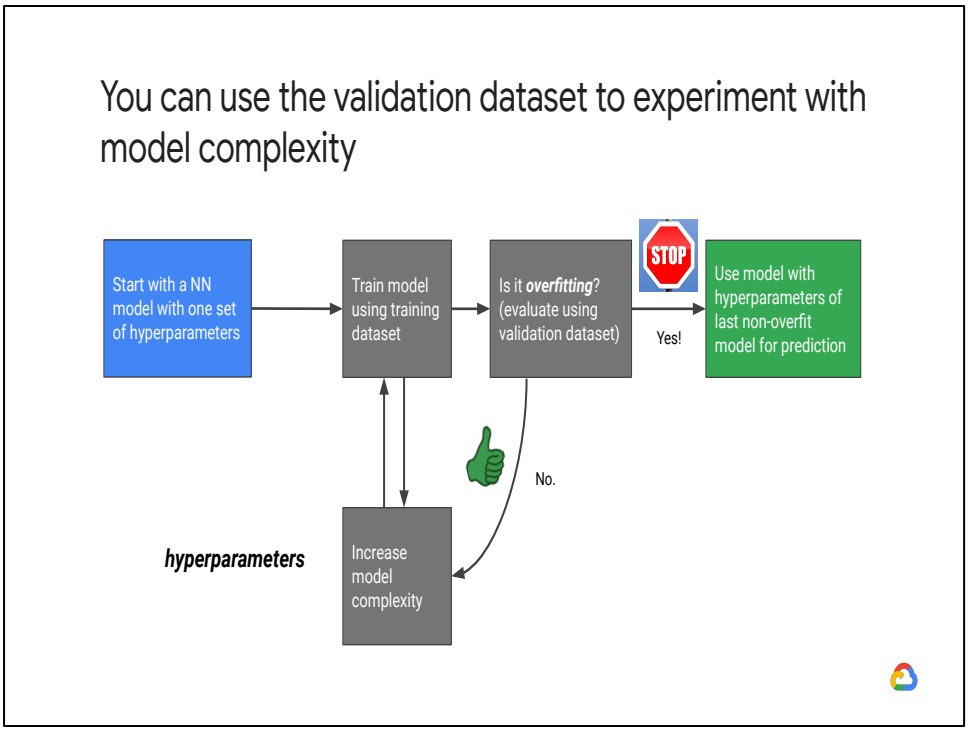
Training and evaluating an ML model is an experiment with finding the right generalizable model that fits your training dataset but doesn't memorize it. As you see here, we have an overly simplistic linear model that doesn't fit the relationships in the data. You'll be able to see how bad this is immediately by looking at your loss metric during training (and visually on this graph here as there are quite a few points outside the shape of trend line). This is called underfitting.

On the opposite end of the spectrum is overfitting, as shown on the right extreme. Here we greatly increased the complexity of our linear model and turned it into an n-th order polynomial which seems to model the training dataset really well -- almost too well. This is where the evaluation dataset comes in -- you can use the evaluation dataset to determine if the model parameters are leading to overfitting. Overfitting or memorizing your training dataset can be far worse than having a model that only adequately fits your data.

Somewhere in between an underfit where the loss metric is not low enough and an overfit whether the model doesn't generalize is the right level of model complexity.

Let's look at how we can use our evaluation dataset to help us know when to

stop training to prevent overfitting.



In addition to helping you choose between two completely different ML models (like regression or neural networks), you can also use your validation dataset to help fine-tune the hyperparameters of a single model which, if you recall, are set before training. This tuning process is accomplished through successive training runs and comparisons against your validation dataset to check for overfitting.

So here's how your validation dataset will actually be used after your model training. As you saw when we covered Optimization, training the model is where we start with random weights, calculate a loss metric on a batch of data, adjust the weights to minimize the loss metric, and repeat.

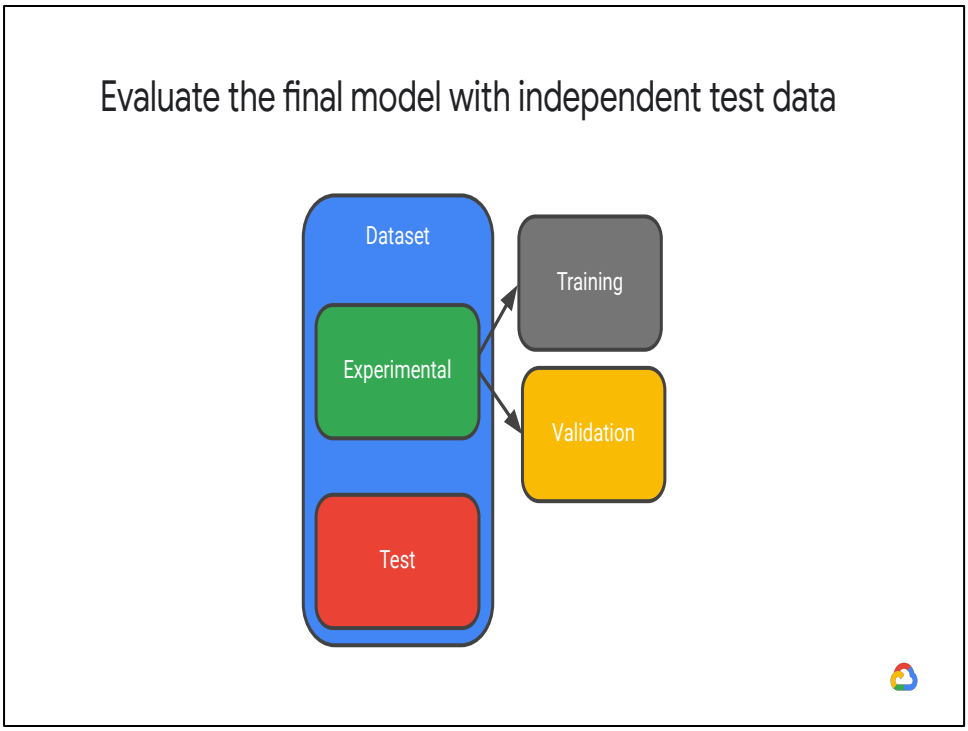
Periodically, we want to assess the performance of our model against data it has not yet seen in training which is where we use our validation dataset after a completed training run with the initial hyperparameters we set.

If there is not significant divergence between the loss metrics from the training run and the loss metric for the validation dataset then the model could still be optimized and tuned. We can go back to our hyperparameters that are set before training happens and try another training model run.
Perhaps we add one more layer to our neural network.

You can use a loop similar to this to also figure out model parameters like what we just did for hyperparameters, for example the number of layers or nodes to use in a neural network. Essentially, you will train with one configuration and evaluate on the validation dataset. Then, you will try out a different configuration that has more or fewer nodes and evaluate on the validation dataset. You will choose the model configuration that results in the lower loss on the validation dataset, not the model configuration that results in lower loss on the training one.

Later in this specialization, we will show you how Cloud ML Engine can carry out a Bayesian search through hyperparameter space, so you don't have to do this kind of experimentation one-hyperparameter-at-a-time. Cloud ML Engine can help us do this sort of experimentation in parallel using a better strategy.

Image cc0 (stop):
https://pixabay.com/en/stop-sign-vector-inkscape-traffic-2717058/



Now, once you are done training, you need to tell your boss how well your model is doing. What dataset do you use to do that final go or no-go evaluation?

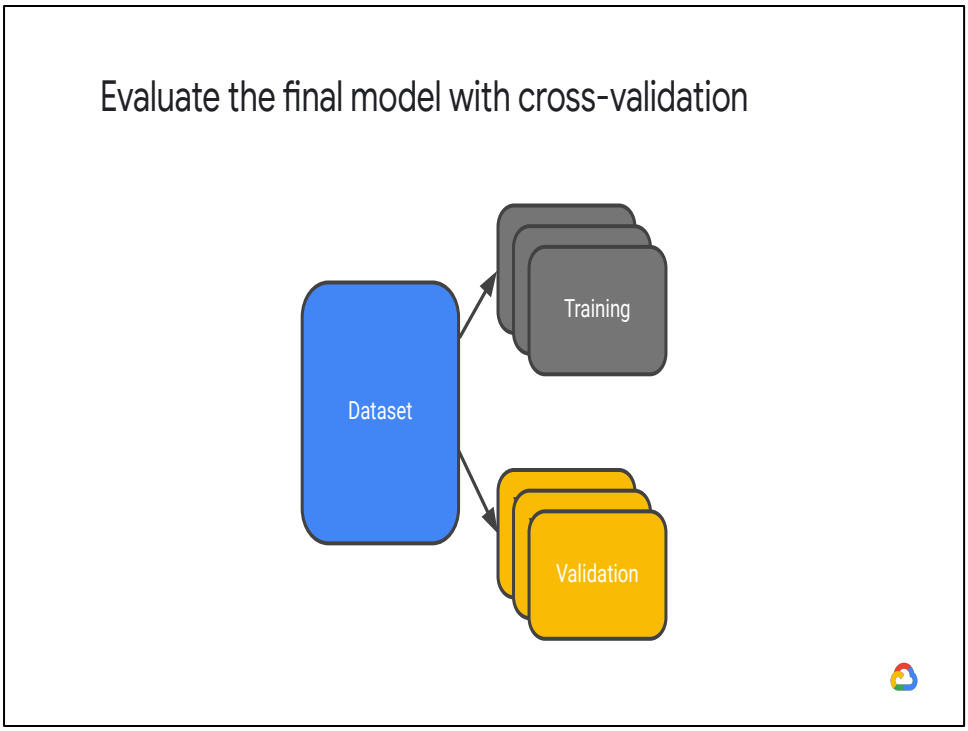
Can you report the error on the validation dataset? Even if it's consistent with your training dataset?

Actually you cant'! Why not? Because you used the validation dataset to choose when to stop the training, remember? *It is no longer independent.*

So, what do you do have to do? Well, you actually have to split your data into into three parts: Training, Validation, and a new completely isolated silo called Test. Once your model has been trained and validated, you can then run it once against the independent Test dataset -- that is the loss metric you report to your boss. It's the loss metric you use to decide whether to use this model.

Now what happens if your model fails to perform against the Test dataset even though it passed validation? It means you cannot re-test the same ML model and need to either (1) create and train a brand new model, or (2) collect more data samples into your original dataset. It's far better for your model to fail in the final test than to fail after you have productionalized it.

While this is a good approach, there is one teeny tiny problem. No one likes to waste data. And it seems that the test data is essentially wasted. It is held out. Can you use all your data in training and still get a reasonable indication of how well your model will perform?



The compromise method is to do the training-validation split many, many times.

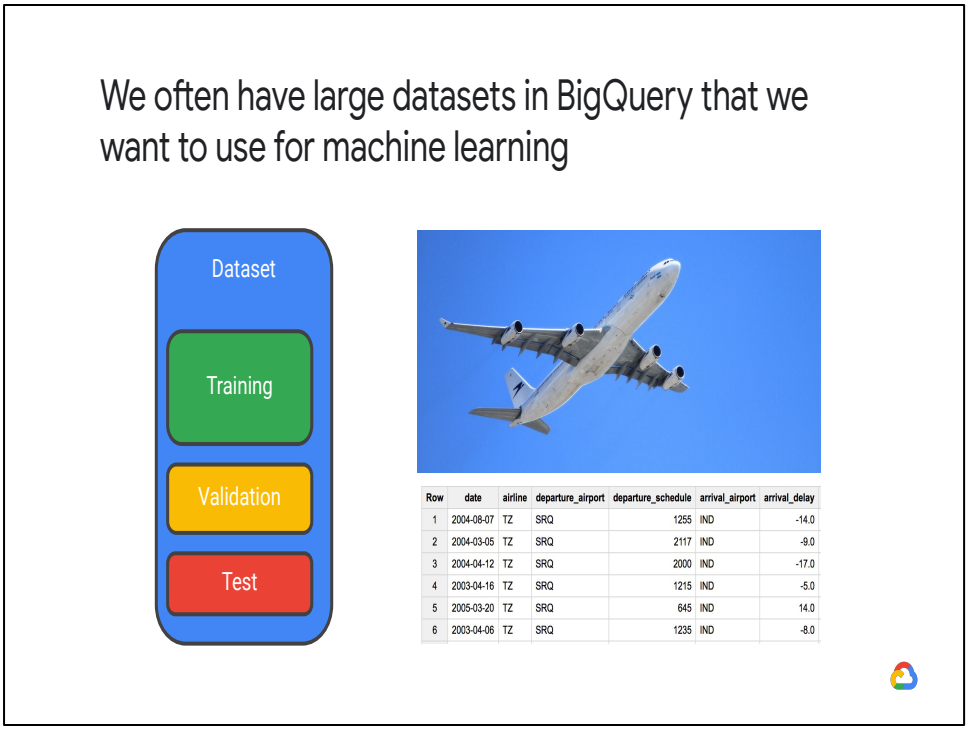
Train and then compute the loss on the validation dataset keeping in mind this validation dataset consists of points that were not used in the training this time. Then, split the data again. Now, your training data might include points that used to be in validation the previous round. Train again and then compute the validation loss metric a second time. Finally after a few rounds of this, *average* all the validation loss metrics. You will also get the standard deviation of the validation loss metrics which will give you a spread to analyze.

This process is called bootstrapping or cross-validation. The upside is that you get to use all the data, but you have to train lots more times.

So here's what you have to remember:
If you have lots of data, use the approach of having a completely independent, held-out test dataset.
If you don't have that much data, use the cross-validation approach.

So how do you go about actuallying splitting large datasets into these silos? That's the topic for our next lesson on sampling.

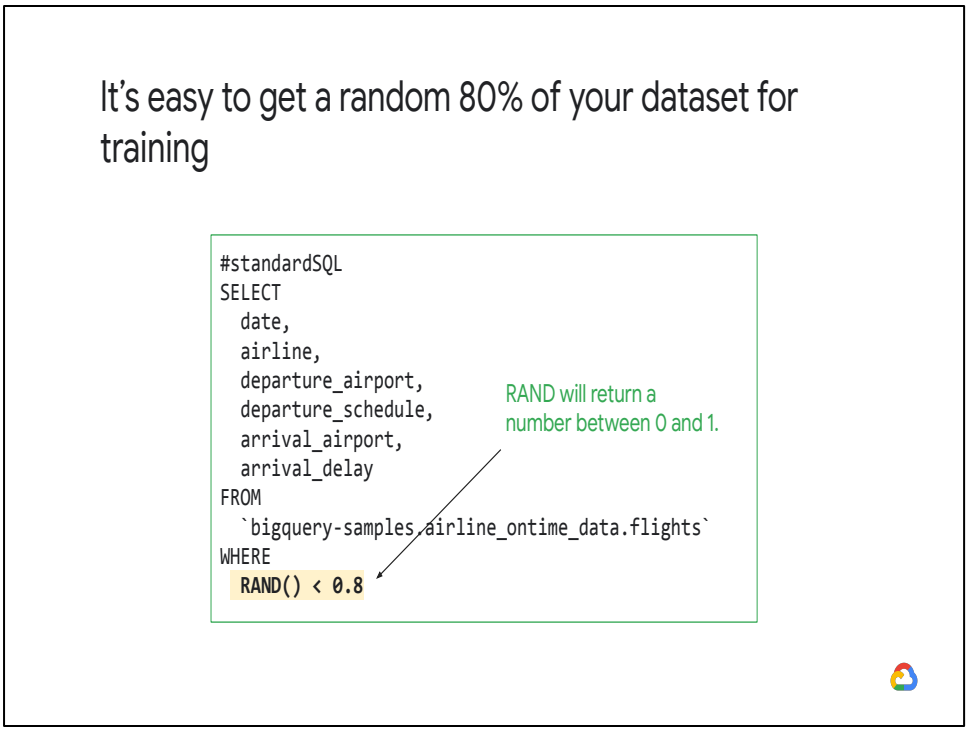
As you just learned, splitting your dataset allows for testing your model against simulated real-world data by holding out subsets of data from training. But how do we know how and where to actually divide our original dataset? What if the dataset itself is extremely large? Do we need to train and test across every data point?



Before we discuss splitting our dataset we first need one to split. For this example, we'll use Airline Ontime Performance data from the U.S. Bureau of Transportation statistics. Google has made this public data available to all users in BigQuery as the airline_ontime_data.flights data set. This dataset has tracked arrival and departure delays for over 70 million U.S. flights.

Let's discuss how we effectively sample training, validation, and testing data from this dataset in a uniform and repeatable way.

Image cc0: https://pixabay.com/en/plane-aircraft-take-off-sky-50893/



SQL (and therefore BigQuery) has the function RAND(), which will generate a value between 0 and 1. You can very easily get 80% of your dataset by just applying a SQL WHERE clause filter as shown here.

You might notice some obvious issues with this… think about whether this process would be repeatable if a colleague wanted to repeat your experiment on the same 80% training dataset. Assuming the dataset was 70 million flights, would they get the same set of 56 million flights (80%) as a training dataset that you did?

However, experimentation requires repeatability

You need to know which specific data was involved in training, validation, and testing.

We need a better way of knowing which data belongs to which bucket: training, validation, and testing. This will enable us and our colleagues to repeat our experiments.

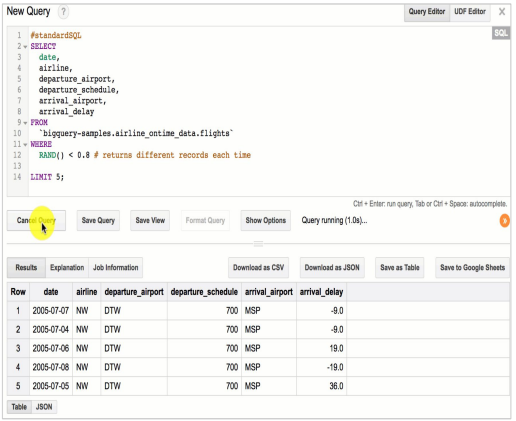
Image cc0: https://pixabay.com/en/testing-experiment-science-test-2681674/

Naive random splitting is not repeatable

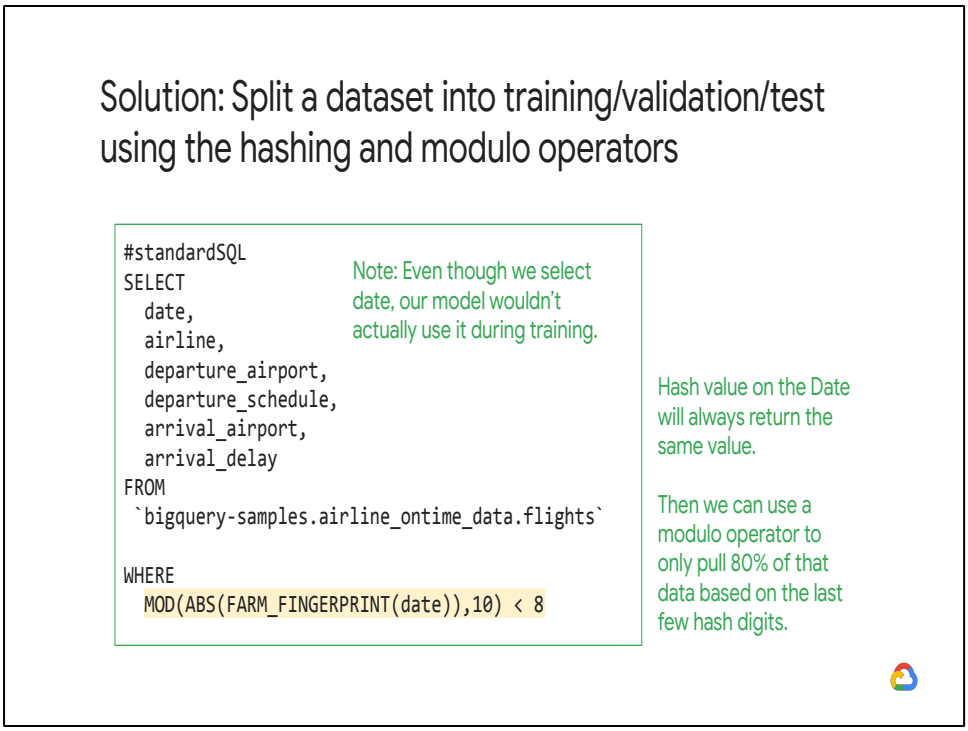
Order of rows in BigQuery is not certain without ORDER BY.

Hard to identify and split the remaining 20% of data for validation and testing.

RAND() will return different results each time →



As you might have guessed, a simple random function will just grab a new set of 5 randomly selected rows here each time you run the query. This makes it extremely difficult to identify and split the remaining 20% of data for validation and testing. In addition, the dataset might be sorted already, which could add bias into your sample (and adding an "order by" comes with its own problems when doing minibatch gradient descent).



For machine learning, you want to be able to repeatedly sample the data you have in BigQuery. One way to achieve this is to use the last few digits of the HASH function on the field that you are using to split your data. One such hash function available in BigQuery is FARM_FINGERPRINT. FARM_FINGERPRINT will take a value like “December 10th 2018” and turn it into a long hash of numbers. This hash value will be identical for every other “December 10th, 2018” value in the dataset.

Now let’s say that you are building a machine learning algorithm to predict arrival delays. You might want to **split up your data by *date*** and get approximately 80% of the days in the training data set:

This is now repeatable; because the FARM_FINGERPRINT function returns the same value any time it is invoked on a specific date, you can be sure you will get the **exact same** 80% of data each time. If you want to split your data by arrival_airport (so that 80% of airports are in the training data set), compute the FARM_FINGERPRINT on arrival_airport instead of date.

Looking at the query here, how would you get a new 10% data sample for evaluation?

Change the (less than) < 8 in the query above to (equals) == 8, and for testing data, change it to == 9. This way, you get 10% of samples in evaluation and 10% in testing.

Carefully choose which field will split your data

We hypothesize that flight delay depends on the carrier, time of day, weather, and airport characteristics (# of runways, etc.) We want to predict flight delays. What field should we split our data on?

- Hash on date?
- Hash on airport?
- Hash on carrier name?

Split your data on a field you can afford to lose.

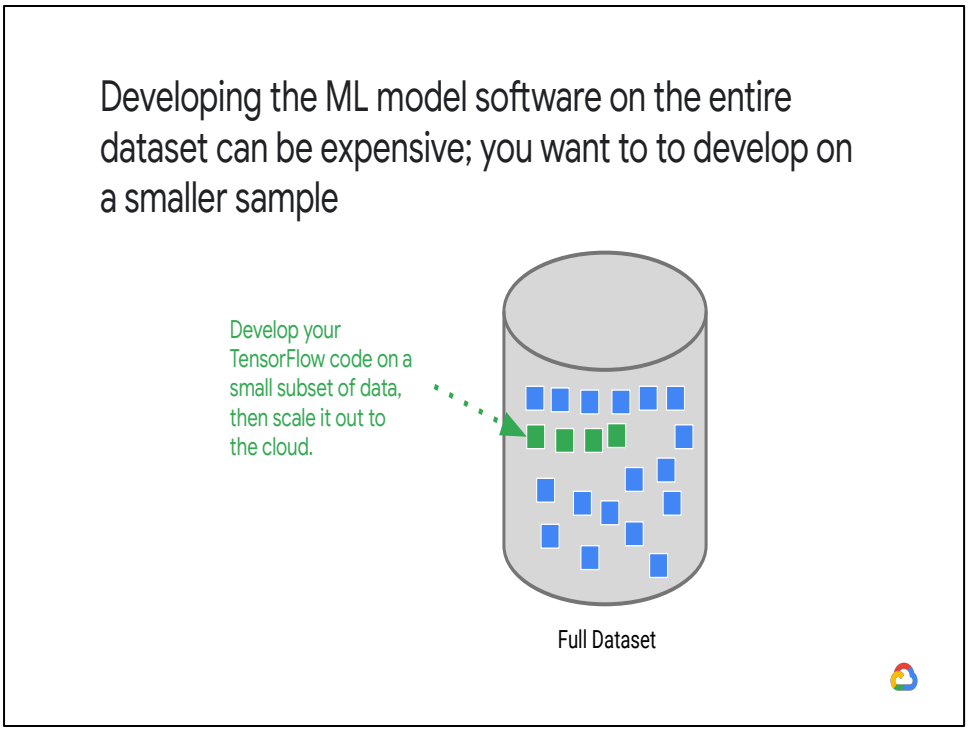
So say we wanted to predict flight delays based on air carrier, time of day, weather, and airport characteristics (# of runways). Which field should we split our dataset on? Date? Airport? Carrier name?

Be sure to split your data into train/valid/test sets based on a column you can afford to lose. The column should also be well distributed and quite noisy. As an example, if you're looking to split on date to predict arrival delays and your dataset only had flights for two days you wouldn't be able to split it more granular than 50% / 50% because the date column isn't distributed enough.

So, let's look at each of these options one-by-one.

What if we Hash and split on Date?
Sure - but understand that you can no longer make predictions based on something like "Christmas," "Thanksgiving,". Be sure the primary drivers in your prediction have nothing to do with date

What if we Hash and split on Airport Name?
Sure - but understand that you can no longer make predictions that are airport-specific; for example, that flights out of JFK at 5 pm are always late.



Starting out with ML model development, it's best to develop TensorFlow code on a small subset of data and then later scale it out on the cloud for productionalization.

Imagine you're developing a ML application. And every time you make a change, you have to run the application.
If you use the full dataset, this can take hours, even days! You can't develop software that way.

You want a small dataset so that you can quickly run your code, and debug it, etc.
Then, once the application is working properly, you can run it on the full dataset.

Next, let's see how we can uniformly sample a smaller subset of our airline dataset.

Pitfall: Chaining hashes to create subsets won't work

```
#standardSQL
SELECT
  date,
  airline,
  departure_airport,
  departure_schedule,
  arrival_airport,
  arrival_delay
FROM
 `bigquery-samples.airline_ontime_data.flights`
WHERE
  MOD(ABS(FARM_FINGERPRINT(date)),70) = 0
    AND
  MOD(ABS(FARM_FINGERPRINT(date)),10) < 8
```

Then take 1 in 70 flights.

Take 80% of the dataset? Incorrect!

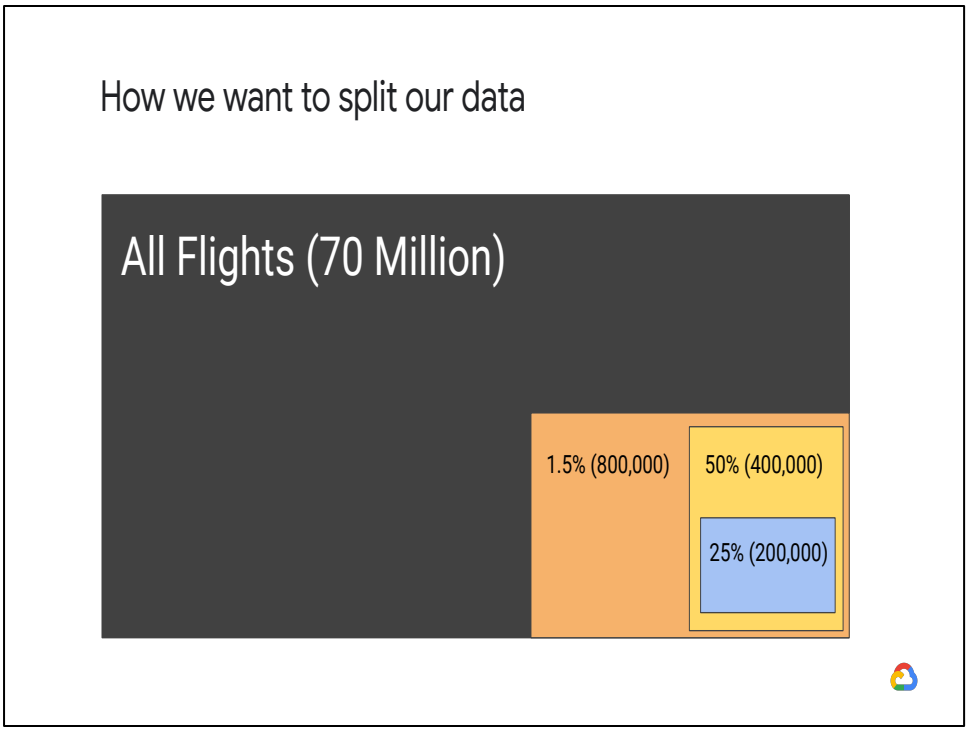
All records here will also be divisible by 10 (there is no new filtering happening!)

What if you want a smaller data set than what you have in BigQuery to play around with? The flight data is 70 million rows, and perhaps what you want is a small data set of 1 million flights. How would you pick one in 70 flights, and then 80% of those as training?

You cannot pick one in 70 rows and then pick one in 10. Can you figure out why?

Well, if you are picking numbers that are divisible by 70, of course they are also going to be divisible by 10! That second modulo operation is useless.

Image cc0: https://pixabay.com/en/sign-road-road-sign-traffic-2460206/



Walk through this query, notes are in comments:
https://bigquery.cloud.google.com/savedquery/133415875420:c0ba13eae1a145a9921521123f40bb76

We can extend this to creating 3 splits

```
#standardSQL
SELECT
  date,
  airline,
  departure_airport,
  departure_schedule,
  arrival_airport,
  arrival_delay
FROM
 `bigquery-samples.airline_ontime_data.flights`
WHERE
  MOD(ABS(FARM_FINGERPRINT(date)),70) = 0
     AND
  MOD(ABS(FARM_FINGERPRINT(date)),700) >= 350
    AND
  MOD(ABS(FARM_FINGERPRINT(date)),700) < 525
```

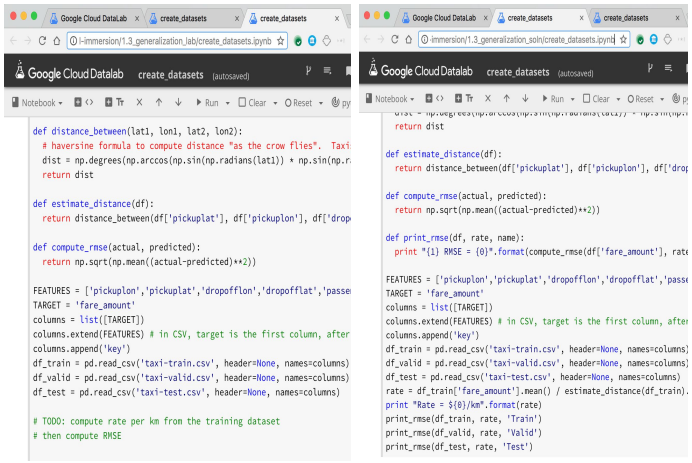
Then take 1 in 70 flights.

Ignore the 50% of the dataset (training).

Choose data between 350 and 524 which is a new 25% sample for Validation.

(this is covered in the previous demo, notes are in public query below)
https://bigquery.cloud.google.com/savedquery/133415875420:c0ba13eae1a145a9921521123f40bb76

Solutions to all the labs exist in the repository



The solutions to all of the labs are in the code repository in GitHub.

This is all open-source. You will have access to the repository even after you finish this course.

Please feel free to use the code as a starting point for your future ML projects.

Benchmarks are important to know what error metric is “reasonable” and/or “great” for the problem

The benchmark helps you set a goal for a good value for the error metric.

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

What’s a good benchmark for the taxi fare prediction?

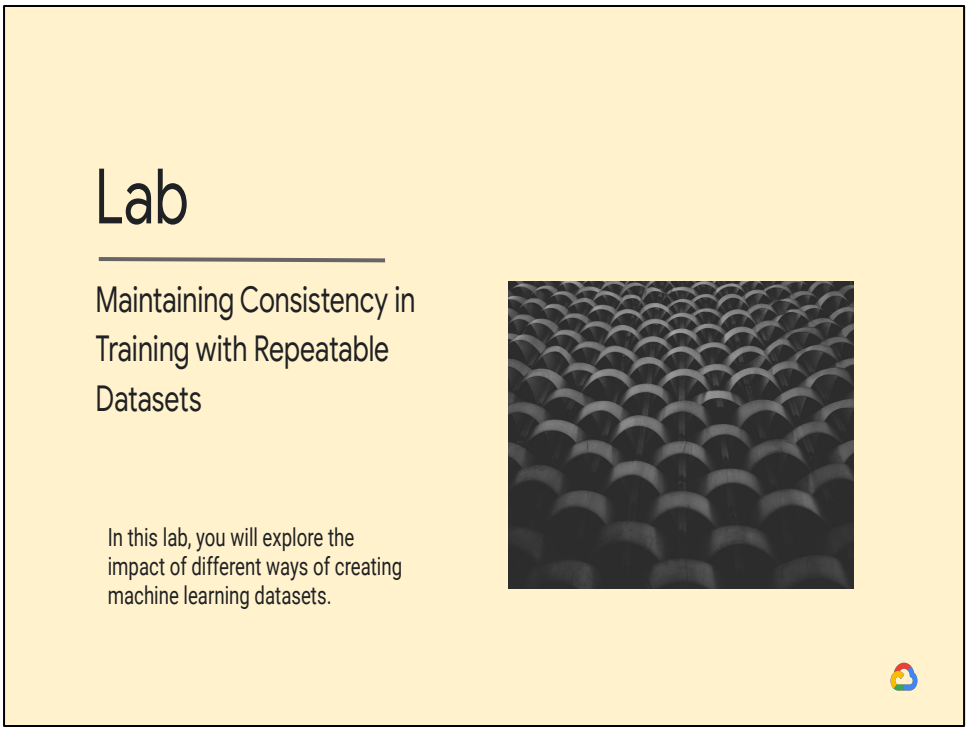
A benchmark is a simple algorithm. Take an RMSE of $3 … is that good or not? You have to have something to compare it to. Having well thought-out benchmarks is a critical step in ML performance and helps us determine whether a model is good enough.

So, what’s a good benchmark for the taxi fare prediction?

Perhaps we could compute the distance between the pickup point and the drop off point, multiply it by a cost-per-kilometer and we are done?

Sure, that seems to be a reasonable benchmark to beat.

Image cc0: https://pixabay.com/en/taxi-cab-traffic-cab-new-york-381233/



In this lab, you will explore the impact of different ways of creating machine learning datasets.

Repeatability is important in machine learning. Imagine you started making changes in your model and meanwhile, the underlying data on subsequent training runs is also changing ... you'll be unable to tell whether your better-performing model is the result of your tweaks or actually because the snapshot of data it used for that run was easier.

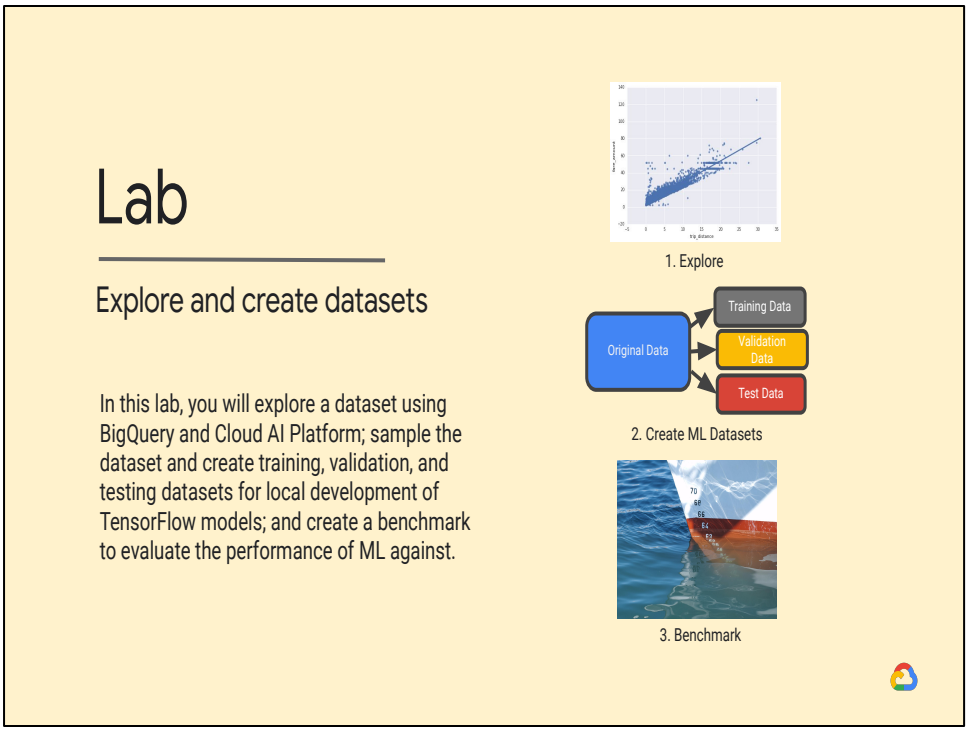
By keeping the data constant as we change the model, we can tweak and tune our model and run it against the exact same experimentation dataset.

You'll practice how to create, split, and hold these datasets constant in your next lab on creating repeatable splits.

Image source: https://pixabay.com/en/pattern-repeating-decorative-1031220/

Link to Lab:

Maintaining Consistency in Training with Repeatable splits github repo link here.



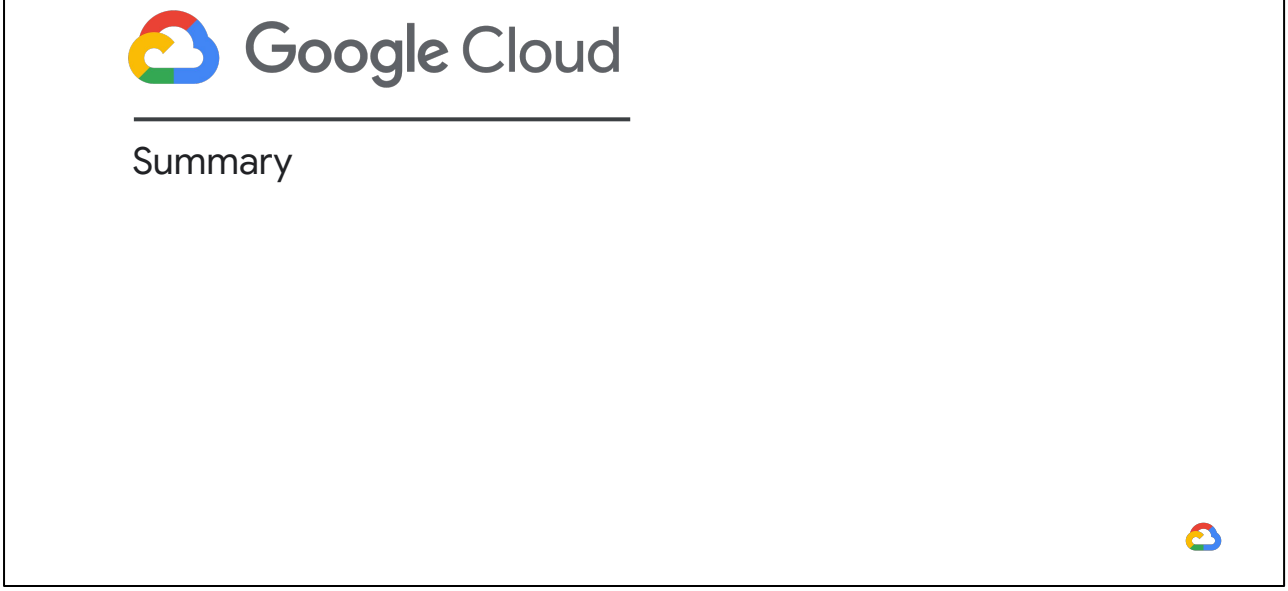
These three steps should form the first steps for any ML project that you undertake. You will often spend weeks just exploring the dataset to gain intuition into the problem -- this is crucial to creating good ML models.

This benchmarking phase should not be neglected; if you don't have a benchmark, you won't know what kind of performance you should seek to attain. Many times, errors can be detected simply by realizing that the performance of the ML model is nowhere near the benchmark.

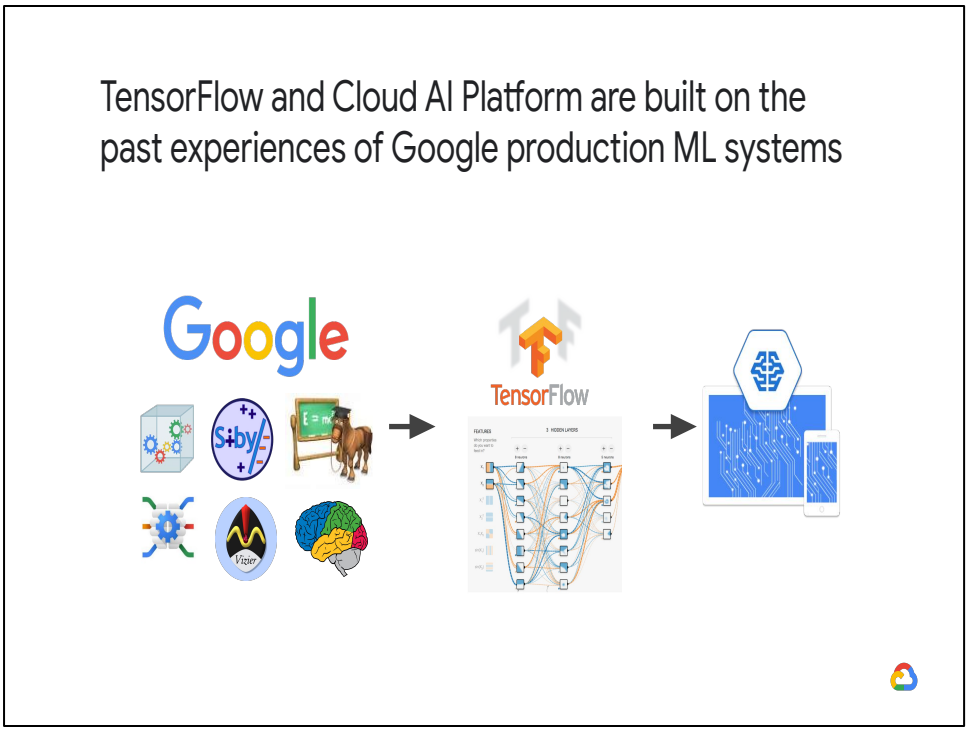
Go ahead and get started with the lab and check your work against the solution and walkthrough when you're ready.

https://pixabay.com/en/ship-sea-water-blue-water-level-1518522/ (cc0)

INSTRUCTOR: THIS LAB IS NOT IN QWIKLABS:
Here is the link to the lab in the repo:
Lab you need to do manually: Explore and Create ML Datasets

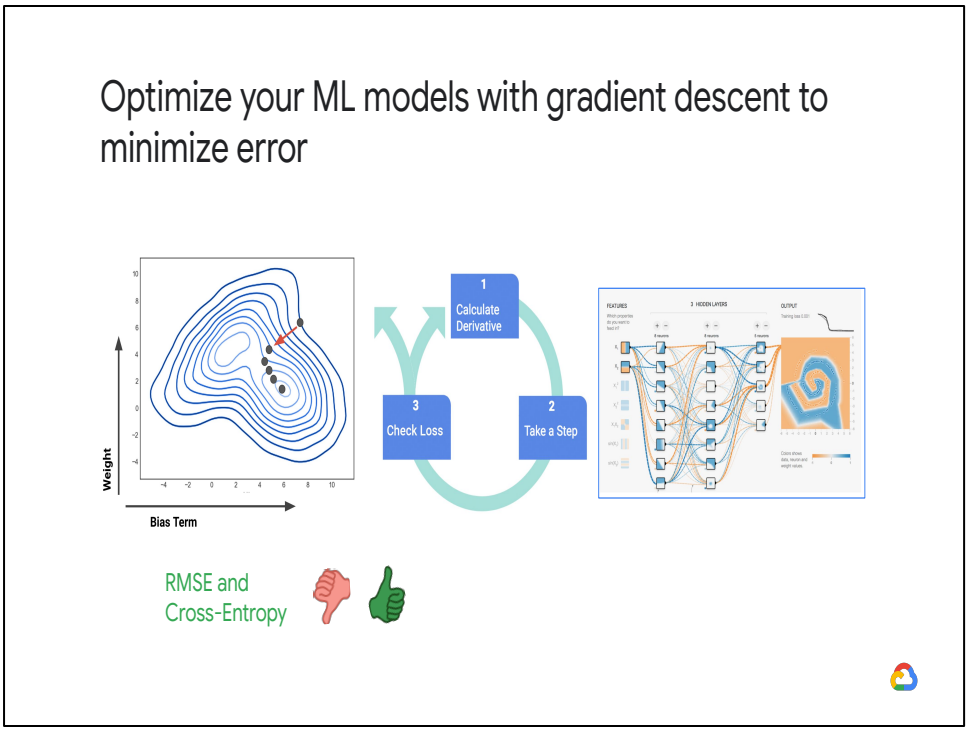


Congratulations! You've made it to the end of the Launching into ML course. Let's recap what we have learned so far.



Data is the first step in the ML process. We looked at data and how important it is to improve data quality - as well as perform exploratory data analysis. We looked at how Google production systems are informed from experience. We then walked through the historical timeline of ML and the growth in prominence of Deep Neural Networks—and why they are the best choice in a large variety of problems. Finally we covered how TensorFlow and Cloud AI Platform build on the experience of Google creating all of these systems.

Notes:
TFX: A TensorFlow-Based Production-Scale Machine Learning Platform
https://dl.acm.org/citation.cfm?id=3098021

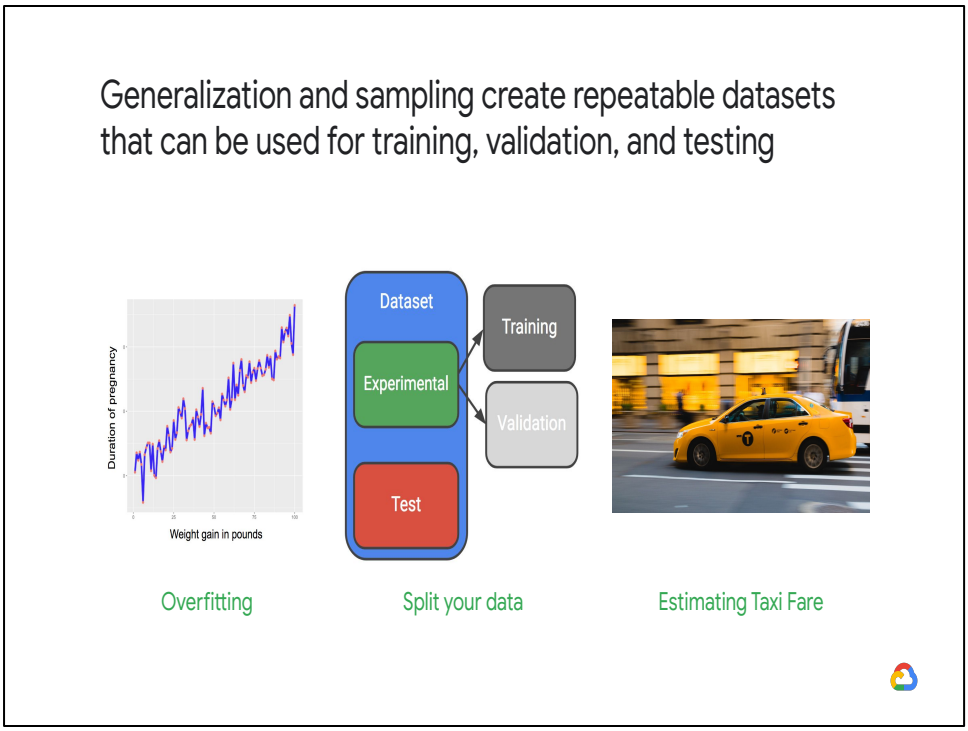


Next we searched through parameter space to find the optimal ML model by using gradient descent down our Loss surfaces. Here we illustrated model training by taking the derivative of the loss surface as our guide towards a minima (keeping in mind there could be more than one for complex surfaces). The Gradient descent process is an iterative one. The idea is to change the weights of the model slightly and reevaluate and use the direction of maximum change, the gradient, to decide which direction to change the weights.

We introduced multiple loss functions like RMSE for Regression problems and Cross-Entropy for classification; introduced performance measures like accuracy, precision, and recall; and discussed the pros and cons of reporting with each.

We experimented in the TensorFlow Playground with how a low, moderate, and very high learning rate affects the shape of our loss curve and can lead to inconsistent model output weights if the learning rate is too high.

We concluded the Optimization module by training neural networks to classify data points in a spiral and ended up with a seemingly complex set of nodes and hidden layers. To better understand whether that model will perform well in the real world is where we headed to next in Generalization.



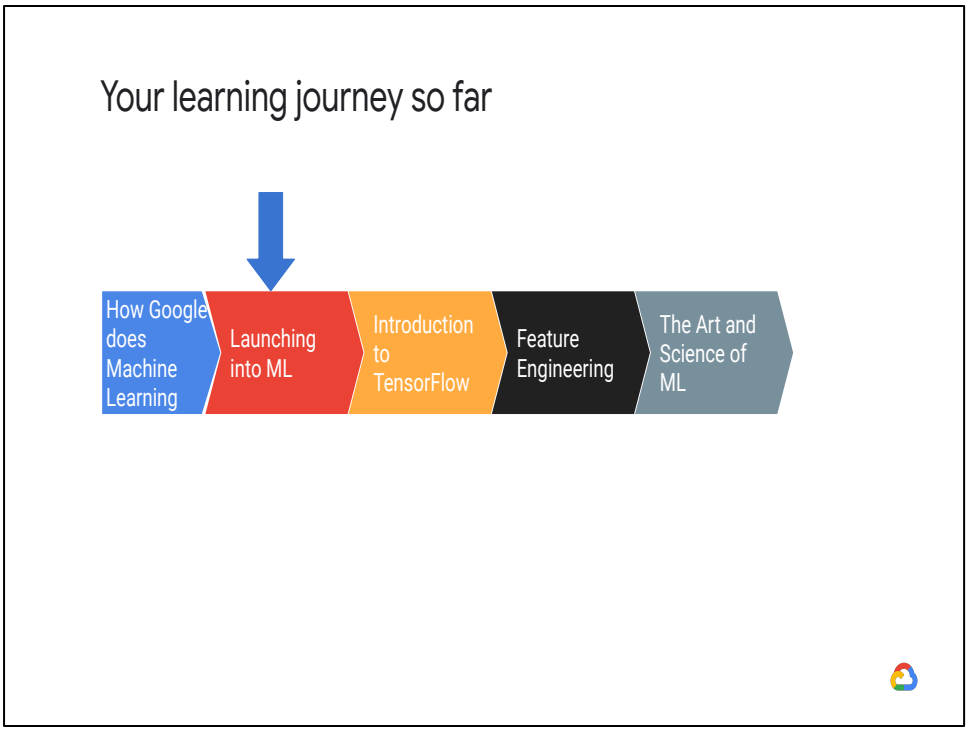
Once we had the perfectly accurate model with RMSE of 0 we then saw how badly it performed against a set of new data it had not seen before. To make our models generalize well and not simply memorize the training dataset we split our original dataset into training, evaluation, and testing and only showed them to the model at predefined milestones.

We then discussed how to create these subsets of of data by splitting and sampling our 70 million flight records dataset in a repeatable fashion. This allowed us to experiment with model improvements and keep the underlying data constant during each model run.

In our lab we discovered that ML models can make incorrect predictions for a number of reasons; poor representation of all use cases, overfitting, and underfitting. We also learned that we can measure the quality of our model by examining the predictions it made.

Keep practicing your ML skills in the hands-on labs, and we'll see you in the next course!

Image (taxi) cc0: https://pixabay.com/en/taxi-new-york-yellow-cab-nyc-2729864/



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

Join for the next chapter, which will be an introduction to Tensorflow

cloud.google.com