Machine Learning Modeling Pipelines in Production

In the third course of Machine Learning Engineering for Production Specialization, you will build models for different serving environments; implement tools and techniques to effectively manage your modeling resources and best serve offline and online inference requests; and use analytics tools and performance metrics to address model fairness, explainability issues, and mitigate bottlenecks.

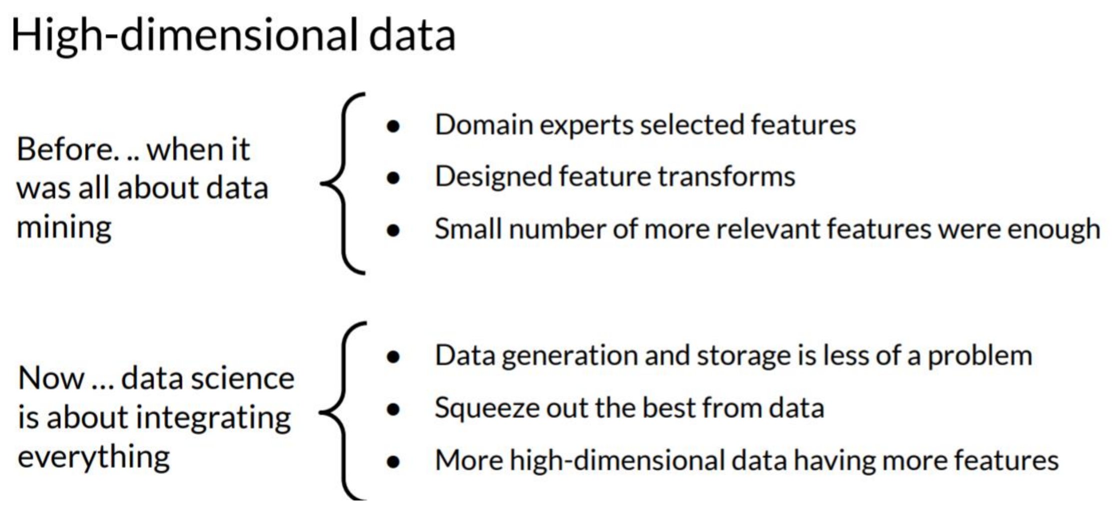
Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise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

Week 2: Model Resource Management Techniques

Contents

Compiled by **Kenneth Leung** Version: **202112**

Dimensionality Reduction



High-dimensional data

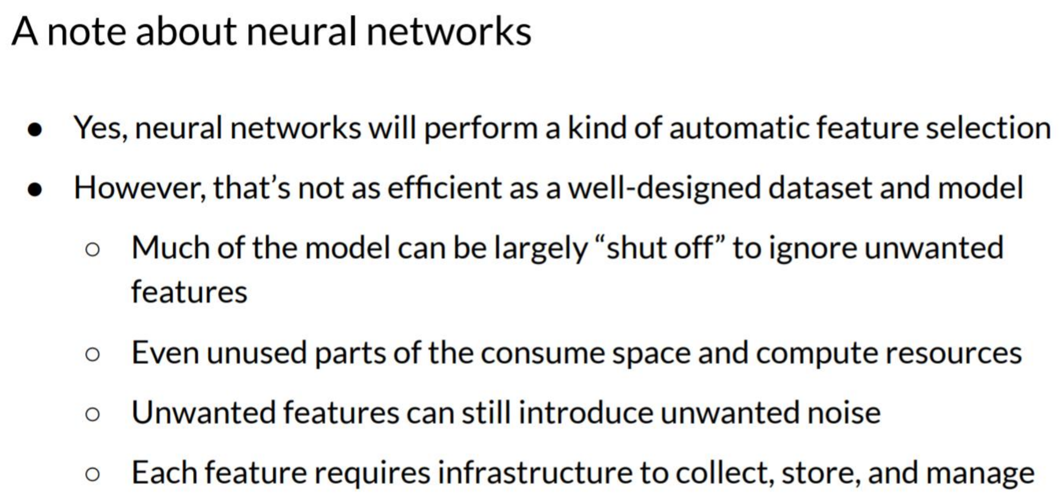
Before. .. when it was all about data mining

- Domain experts selected features
- Designed feature transforms
- Small number of more relevant features were enough

Now … data science is about integrating everything

- Data generation and storage is less of a problem
- Squeeze out the best from data
- More high-dimensional data having more features

- The compute, storage and IOT Systems that your model requires will determine how much it costs to put your model into production and maintain it during its entire lifetime.
- This week we'll be taking a look at some important techniques that can help us manage model resource requirements.
- We'll begin by discussing **dimensionality** and how it affects our models performance and resource requirements.
- In the not so distant past data generation and to some extent data storage was a lot more costly than it is today. Back then a lot of domain experts would carefully consider which features or variables to measure before designing their experiments and feature transforms.
- As a result, data sets were expected to be well designed and potentially contain only a small number of relevant features.
- Today, **data science tends to be more about integrating everything end to end,** generating and storing data is becoming faster, easier and less expensive.
- So there's a tendency for people to measure everything they can and include ever more complex feature transformations.
- As a result, datasets are often high dimensional, containing a large number of features, although the relevancy of each feature for analyzing this data is not always clear.

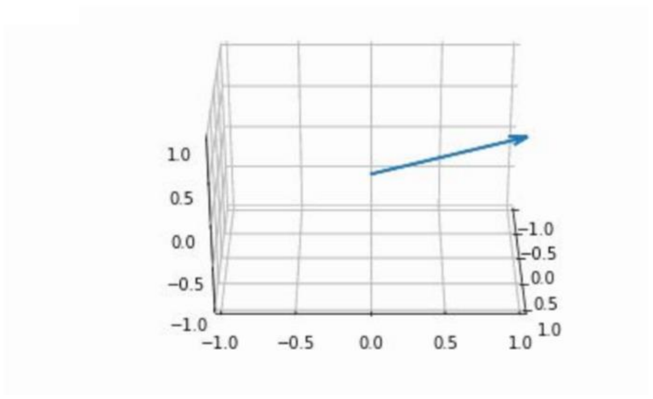


A note about neural networks

- Yes, neural networks will perform a kind of automatic feature selection
- However, that’s not as efficient as a well-designed dataset and model
  - Much of the model can be largely “shut off” to ignore unwanted features
  - Even unused parts of the consume space and compute resources
  - Unwanted features can still introduce unwanted noise
  - Each feature requires infrastructure to collect, store, and manage

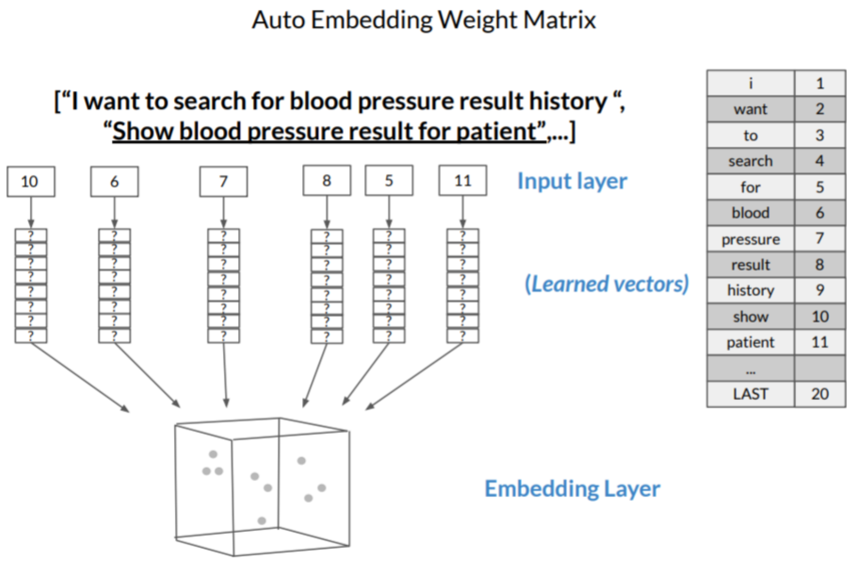
- Before going to deep, let's discuss a common misconception about neural networks. Many developers correctly assume that when they train their neural network models, the model itself, as part of the training process, will learn to **ignore** features that don't provide predictive information by reducing their weights to zero or close to zero.
- Well, while this is true, the result is **not an efficient model**.
- Much of the model can end up being shut off when running inference to generate predictions but those unused parts of the model are still there.
- They **take up space and they consume compute resources** as the model server traverses the computation graph. Those unwanted features could also introduce **unwanted noise** into the data, which can degrade model performance.
- And outside of the model itself each extra feature still requires systems and infrastructure to collect that data, store it, manage updates, etc., which adds **cost and complexity** to the overall system.
- That includes monitoring for problems with the data and the effort to fix those problems if and when they happen.
- Those costs continue for the lifetime of the product or service that you're deploying, which could easily be years.
- There are techniques for **optimizing models with weights that are close to zero**, which you'll learn more about later in this course.
- But **in general, you shouldn't just throw everything at your model** and rely on your training process to determine which features are actually useful.

High-dimensional spaces



- In machine learning we often have to deal with high dimensional data.
- Consider an example where we're dealing with recording 60 different metrics for each of our shoppers, which means we're working in a space with 60 dimensions.
- In other cases, if you're trying to analyze grayscale images that are 50 by 50, you're working in an area with 2500 dimensions. If the dimensions or if the images rather are RGB, the dimensionality increases to 7500 dimensions.
- In this case we have one dimension for each color channel in each pixel of the image.

Word embedding - An example



- Some feature representations such as one hot encoding are problematic for working with text in high dimensional spaces as they tend to produce very sparse representations that do not scale well.
- One way to overcome this is to use an **embedding layer that tokenizes the sentences** and assigns a float value to each word.
- This leads to a more powerful vector representation that respects the timing and sequence of the words in a given sentence. This representation can be automatically learned during training. The cube labeled embedding layer in this figure is a conceptual representation of those vectors in a high dimensional space.

Initialization and loading the dataset

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
from keras.datasets import reuters
from keras.preprocessing import sequence
num_words = 1000

(reuters_train_x, reuters_train_y), (reuters_test_x, reuters_test_y) =
                    tf.keras.datasets.reuters.load_data(num_words=num_words)
n_labels = np.unique(reuters_train_y).shape[0]
```

- Let's look at a concrete example of word of embedding using Keras. Let's start by loading the necessary libraries and modules into finding some important parameters.
- The Reuters news data set that we will be working with contains 11,228 newswires labeled over 46 topics.
- The documents are already encoded in such a way that each word is indexed by an integer, its overall **frequency** in the data set.
- While loading the data set, we specify the number of words we will work with (1000), so that the least repeated words are considered unknown.

Further preprocessing

```
from keras.utils import np_utils
reuters_train_y = np_utils.to_categorical(reuters_train_y, 46)
reuters_test_y = np_utils.to_categorical(reuters_test_y, 46)

reuters_train_x =
     tf.keras.preprocessing.sequence.pad_sequences(reuters_train_x, maxlen=20)
reuters_test_x = tf.keras.preprocessing.sequence.pad_sequences(reuters_test_x,
                 maxlen=20)
```

- Let's further pre-process the data, so it's ready for training a model.
- First this converts the training vector Y into a categorical variable for both train and test. Next, the code segments the input text into 20 word long sequences.

Using all dimensions

```
from tensorflow.keras import layers
model2 = tf.keras.Sequential(
    [
      layers.Embedding(num_words, 1000, input_length= 20),
      layers.Flatten(),
      layers.Dense(256),
      layers.Dropout(0.25),
      layers.Activation('relu'),
      layers.Dense(46),
      layers.Activation('softmax')
    ])
```

- Building the network is the next logical step. So here the choice is to embed a 1000 word vocabulary using all dimensions, here we're using all the dimensions of the data.
- The last layer is dense with dimension 46 since the target variable is a 46 dimensional vector of categories.

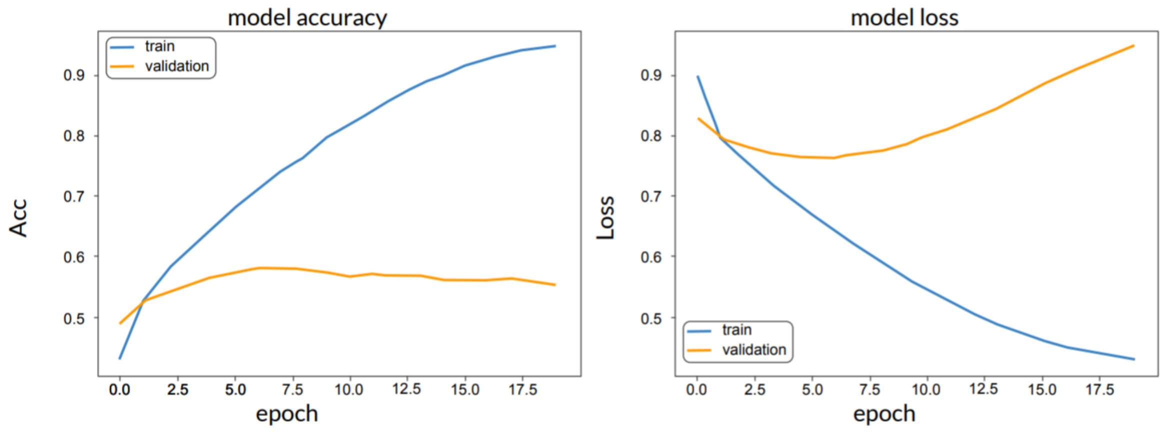
Model compilation and training

```
model.compile(loss="categorical_crossentropy", optimizer="rmsprop",
metrics=['accuracy'])


model_1 = model.fit(reuters_train_x, reuters_train_y,
                    validation_data=(reuters_test_x , reuters_test_y),
                    batch_size=128, epochs=20, verbose=0)
```

- With the model structure ready let's compile the model by specifying the loss optimizer and output metric.

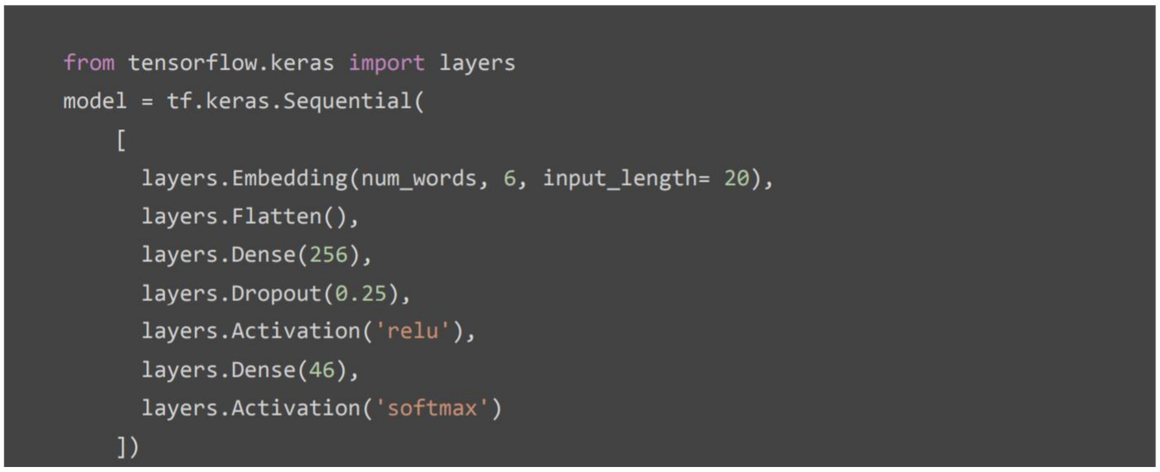
- For this problem the natural choices are categorical crossentropy loss, rmsprop optimization and accuracy is the metric.
- Now all is set to do a model fitting. We will specify the validation set, batch size and the number of epochs for training.

Example with a higher number of dimensions



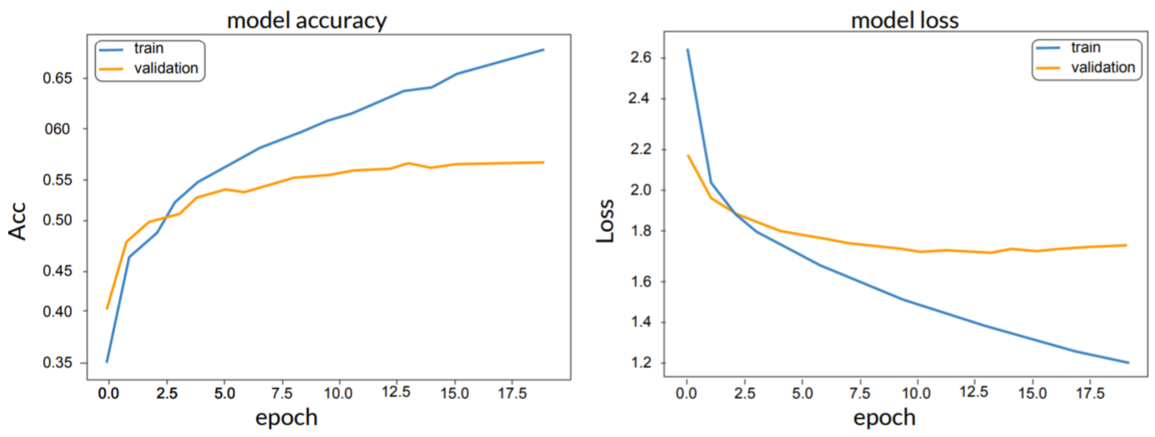
- Here is the accuracy as a function of training epochs and the model loss as a function of training epochs.
- Notice that after about two epics are training data results in significantly higher accuracies and lower losses compared to the validation set.
- This is a clear indication that the model is severely **overfitting**, and this may be the result of using all the dimensions of the data.
- So the model is picking up nuances in the training set that do not generalize well.

Word embeddings: 6 dimensions


```
from tensorflow.keras import layers
model = tf.keras.Sequential(
    [
      layers.Embedding(num_words, 6, input_length= 20),
      layers.Flatten(),
      layers.Dense(256),
      layers.Dropout(0.25),
      layers.Activation('relu'),
      layers.Dense(46),
      layers.Activation('softmax')
    ])
```

- Let's try reducing the dimensionality and see how this affects model performance. For that let's embed a 1000 word vocabulary into 6 dimensions, this is roughly a reduction of a fourth root factor.

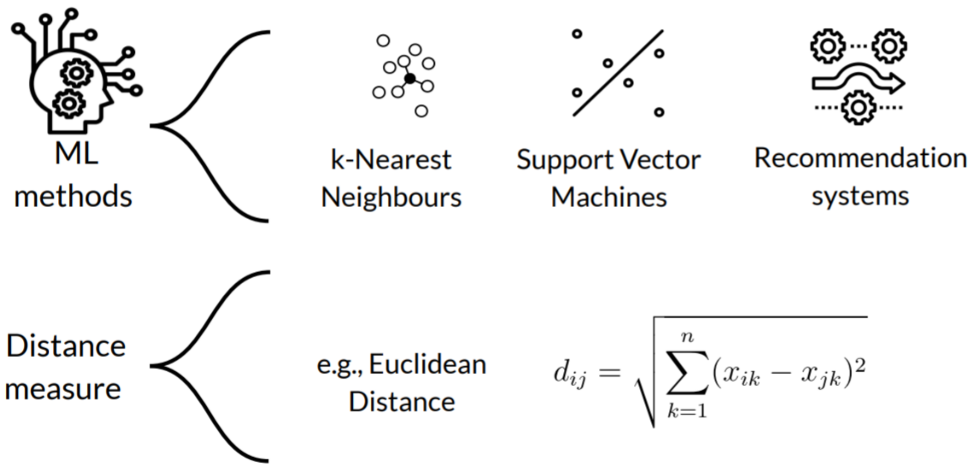
Word embeddings: fourth root of the size of the vocab



- The model remains unchanged otherwise. There may still be some overfitting, but with that one change, this model performs significantly better than the 1000 dimension version.

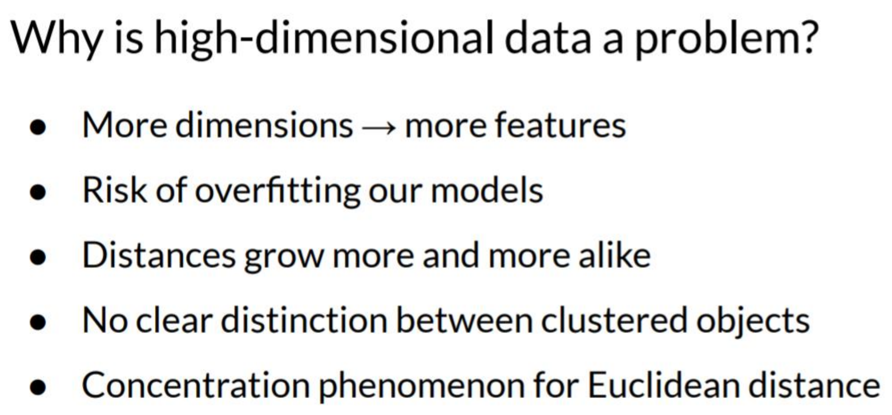
Curse of Dimensionality

Many ML methods use the distance measure



$$d_{ij} = \sqrt{\sum_{k=1}^{n}(x_{ik} - x_{jk})^2}$$

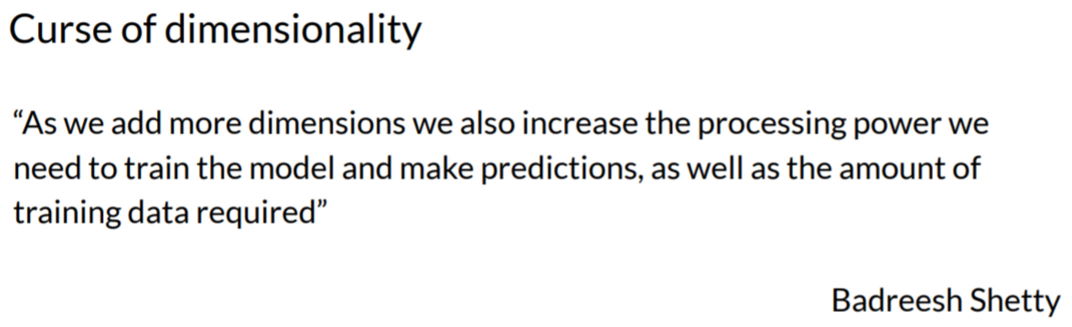
- Let's talk about the curse of dimensionality and why this is a very important topic when building models. –
- Many common machine learning tasks like segmentation and clustering rely on computing distances between observations.
- For example, supervised classification uses the distance between observations to assign a class, k-nearest neighbors is a basic example of this.
- Support vector machines or SVMs deal with projecting observations using kernels based on the distance between the observations after projection.
- Another example is recommendation systems that use a **distance based similarity measure** between the user and the item attribute vectors. There could even be other forms of distance being used.
- So distance plays an important role in understanding dimensionality.
- One of the most common distance metrics is Euclidean distance, which is simply a linear distance between two points in a multi-dimensional space.
- The Euclidean distance between two dimensional vectors with Cartesian coordinates is calculated using this familiar formula.



Why is high-dimensional data a problem?

- More dimensions → more features
- Risk of overfitting our models
- Distances grow more and more alike
- No clear distinction between clustered objects
- Concentration phenomenon for Euclidean distance

- But why is distance important? Let's look at some issues with measuring distance in high dimensional spaces.
- For example you might wonder why data being high dimensional can be an issue.
- In extreme cases where we have more features and observations, we run the **risk of massively overfitting our model**.
- But in the more general case when we have too many features, observations become harder to cluster.
- Too many dimensions can cause every observation in your data set to **appear equidistant** from all the others.
- And because clustering using a distance measures such as Euclidean distance to quantify the similarity between observations, this is a big problem.
- If the distances are all approximately equal, all the observations appear equally alike and **no meaningful clusters** can be formed.
- As dimensionality grows, the contrast provided by the usual metrics decreases. In other words, the distribution of norms in a given distribution of points tends to concentrate. That can cause unexpected behavior in high dimensional spaces.



Curse of dimensionality

"As we add more dimensions we also increase the processing power we need to train the model and make predictions, as well as the amount of training data required"

Badreesh Shetty

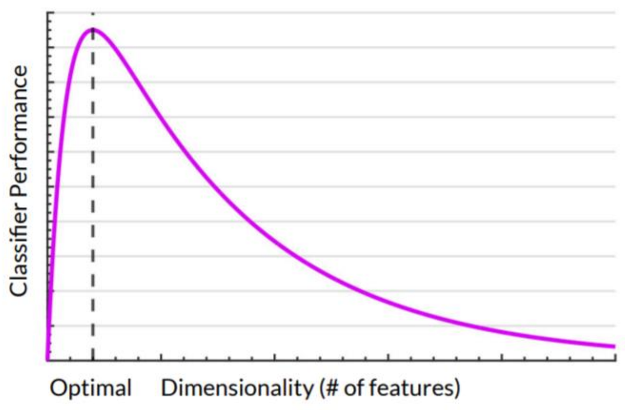
- This phenomenon is called the curse of dimensionality.
- It includes situations where non-intuitive properties of data are observed in these high dimensional spaces. This is specifically related to the usability and interpretation of distances and volumes.
- When it comes to the curse of dimensionality, there are two things to consider.
- On the one hand, machine learning is good at analyzing data when dealing with many dimensions.
- However, **we humans aren't adept** at finding patterns and data that may be spread out across several dimensions, especially if those dimensions are interrelated in counterintuitive ways.
- On the other hand, as we add more dimensions, we also increase the **processing power** we need to analyze the data and at the same time we also increase the amount of training data required to make meaningful models.
- If you're curious, Richard Bellman first coined the term curse of dimensionality over half a century ago in 1961 in his book Adaptive Control Processes, A Guided Tour.

Why are more features bad?

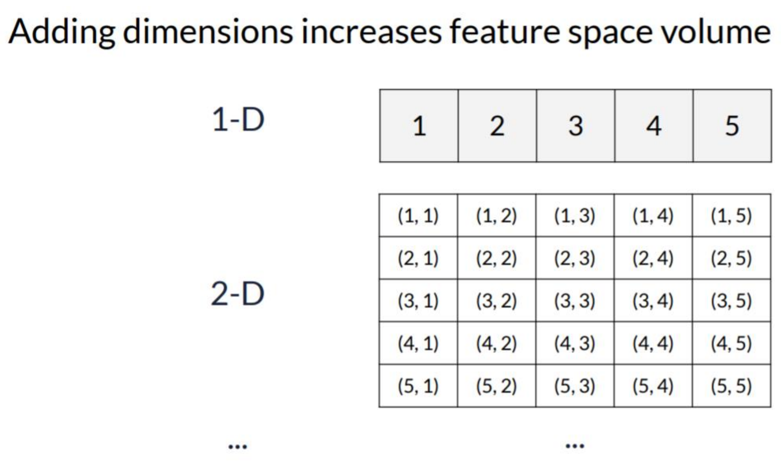
- Redundant / irrelevant features
- More noise added than signal
- Hard to interpret and visualize
- Hard to store and process data

- So adding more features can easily create problems. This could include redundant or irrelevant features appearing in data.
- Moreover, noise is added when features don't provide predictive power for our models.
- On top of that, more features make it harder for one to interpret and visualize data.
- Finally, more features mean more data, so you need to have more storage and more processing power to process it.
- Ultimately, having more dimensions often means our model is less efficient.

The performance of algorithms ~ the number of dimensions



- When you have problems getting your model to perform, you are often tempted to try adding more and more features.
- But as you add more features, you reach a certain point where your model's performance degrades. This graph shows this well.
- Here you see that as the dimensionality increases, the classifiers performance increases until the optimum number of features is reached.
- A key point to understand here is that you are increasing the dimensionality without increasing the number of training samples and that results in a steady decrease in classifier performance after the optimum.
- Let's look at another problem with dimensionality to uncover what is behind this behavior.



Adding dimensions increases feature space volume

1-D

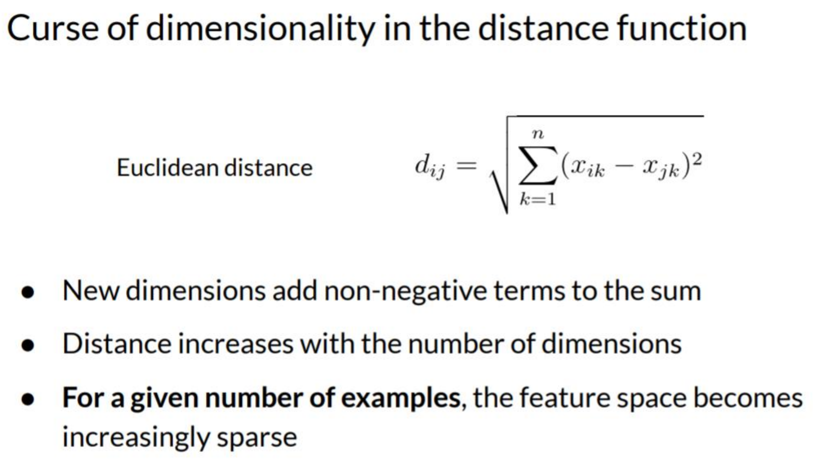
1	2	3	4	5

2-D

(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)
(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)
(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)
(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)

...   ...

- Let's look deeper to try to understand why more dimensions can hurt your models.
- Let's start by understanding why the number of parameters of a function impact the difficulty of learning that function.
- Take for example, the parameters of a line function. In this case the list is finite. It simply means discretizing the features. Let's simplify this by using numbers from 1-5.
- Assuming that you have a function with a single parameter, then there are only five possible values that this parameter can take.
- What happens if you add a second parameter that can also take five values?
- There are now 5 times 5 or 25 possible pairs. This is a simple example using discrete parameter values, but of course it's even worse with continuous variables.
- What happens if you had a third parameter? Now the representation is a cube, you probably see the formula behind this reasoning. If we have n values that a parameter can take and m parameters, you end up with n to the m possible parameter values.
- The number of parameter values grows **exponentially**.
- How big of a problem is it? Well, it's a big problem, which is why this is called the curse of dimensionality.



Curse of dimensionality in the distance function

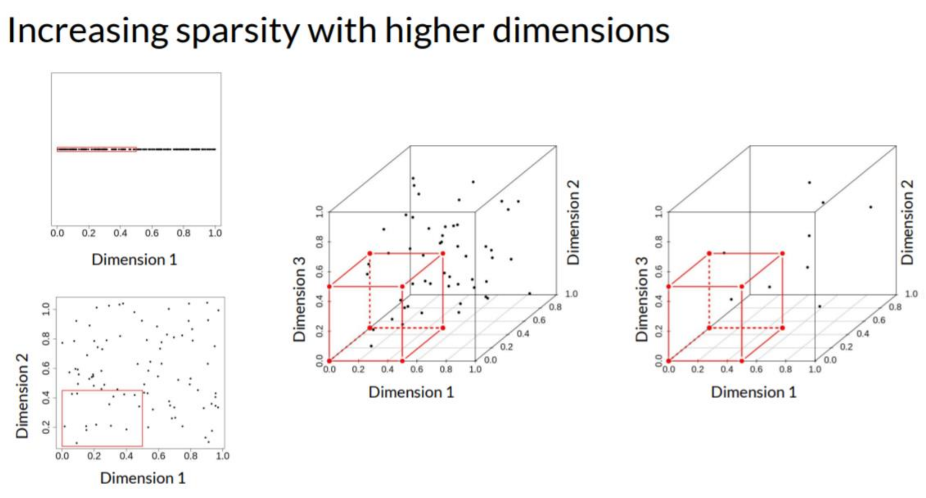
Euclidean distance

$$d_{ij} = \sqrt{\sum_{k=1}^{n}(x_{ik} - x_{jk})^2}$$

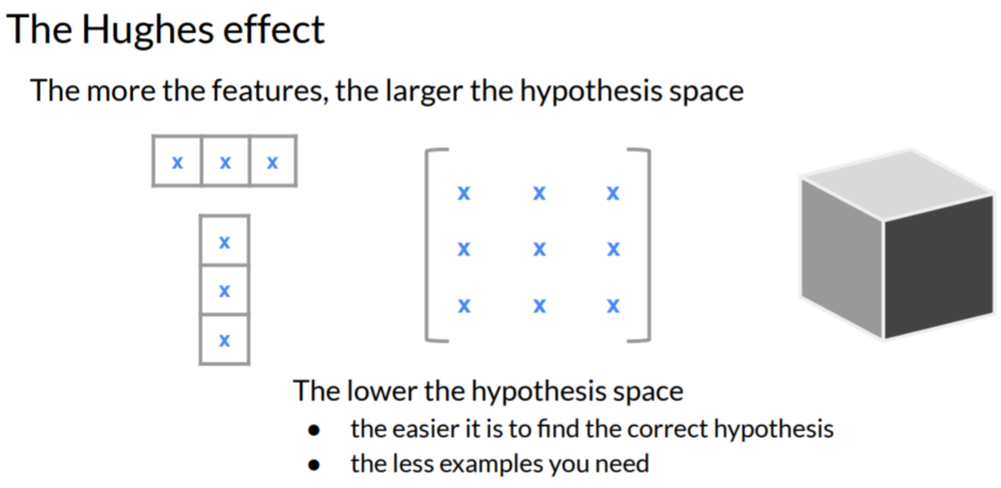
- New dimensions add non-negative terms to the sum
- Distance increases with the number of dimensions
- **For a given number of examples**, the feature space becomes increasingly sparse

- An increase in the number of dimensions of a data set means there are **more entries in the feature vector representing each training example.**
- Let's focus on a Euclidean space and Euclidean distance measure. Each new dimension adds a non-negative term to the sum, so the distance increases with the number of dimensions for distinct factors.
- In other words, as the number of features grows for a given number of training examples, the feature space becomes **increasingly sparse with more distance between training examples**.
- Because of that, the lower data density requires more training examples to keep the average distance between data points the same.

- It's also important that the examples that you add are significantly different from the examples that you already have, or that are already present in the sample.
- Here the argument is built using Euclidean distance, but it is true for any properly defined distance measure.


Increasing sparsity with higher dimensions

- When the distance between observations grows, supervised learning becomes more difficult because predictions for new samples are less likely to be based on learning from similar training examples.
- The size of the feature space grows exponentially as the number of features increases making it much harder to generalize efficiently.
- The variance increases and there's a higher chance of overfitting to noise in more dimensions resulting in poor generalization performance.
- In practice, features can also be correlated or do not exhibit much variation.
- For these reasons, there is a need to reduce dimensionality. The challenge is to keep as much of the predictive information as possible using as few features as possible.


The Hughes effect

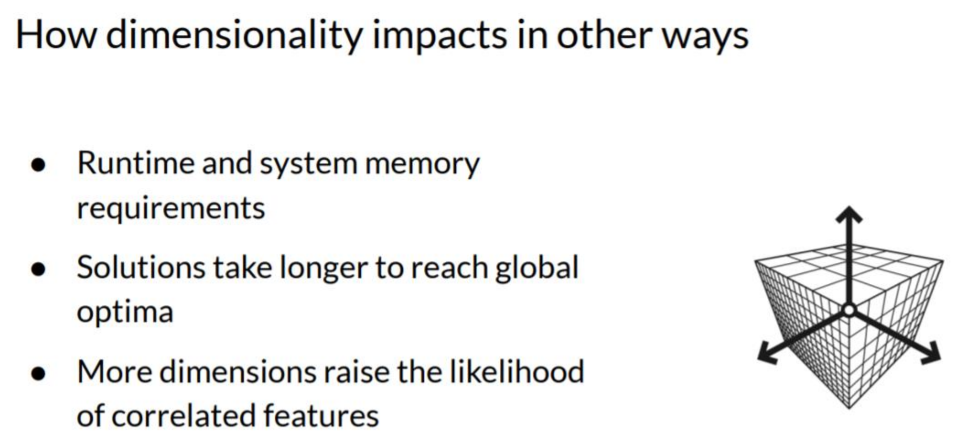
The more the features, the larger the hypothesis space

The lower the hypothesis space
- the easier it is to find the correct hypothesis
- the less examples you need

- Regardless of which modeling approach you're using, increasing dimensionality has another problem especially for classification which is known as the Hughes effect.
- This is a phenomenon that demonstrates the improvement in classification performance as the number of features increases until we reach an optimum where we have enough features.
- Adding more features while keeping the training set the same size will degrade the classifiers performance. We saw this earlier in our graph.
- In classification, the goal is to find a function that discriminates between two or more classes.
- You can do this by searching for hyperplanes in space that separate these categories.

- The more dimensions you have, the easier it is to find a hyperplane during training, but at the same time the harder it is to match that performance when generalizing to unseen data.
- And the less training data you have, the less sure you are that you identify the dimensions that matter for discriminating between categories.

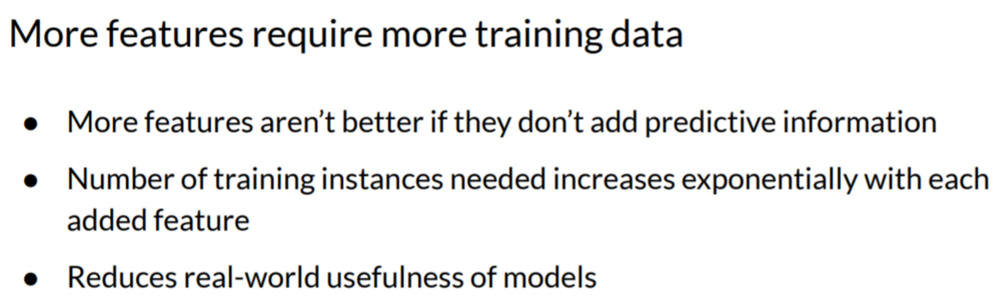
Curse of Dimensionality – An Example



How dimensionality impacts in other ways

- Runtime and system memory requirements
- Solutions take longer to reach global optima
- More dimensions raise the likelihood of correlated features

- Let's look at an example of how dimensionality reduction can help our models perform better
- Apart from distances and volumes, increasing the number of dimensions can create other problems.
- Processor and memory requirements often scale non-linearly with an increase in the number of dimensions, due to an exponential rise in feasible solutions many optimization methods cannot reach a global optima and get stuck in a local optima.
- More dimensions also often increases the likelihood of having **correlated** features and parameter estimation can often be challenging in regression models.
- So let's look at how more features can make training a model harder, with an example

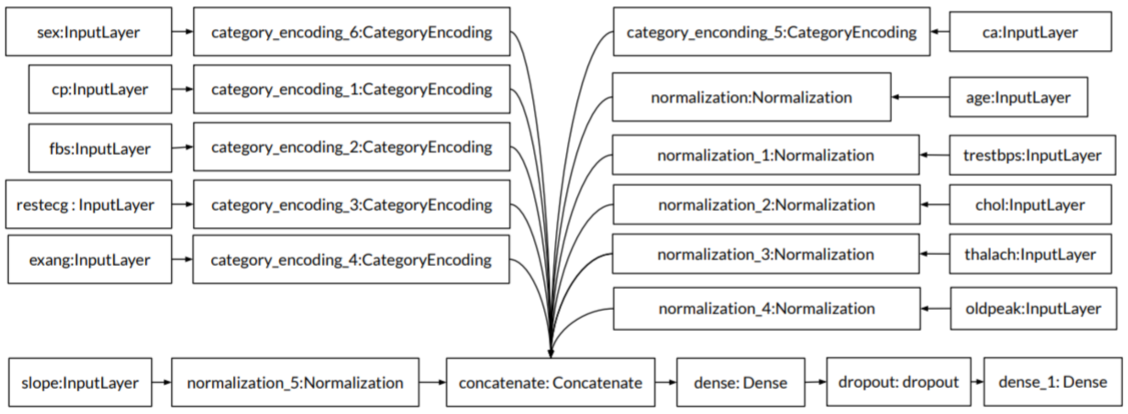


More features require more training data

- More features aren't better if they don't add predictive information
- Number of training instances needed increases exponentially with each added feature
- Reduces real-world usefulness of models

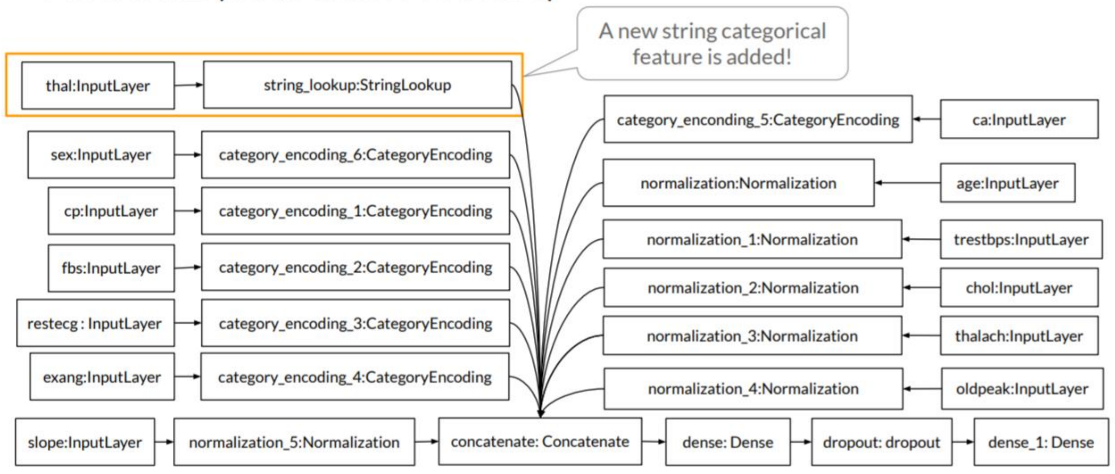
- When you create a model, you design it for a certain number of features or dimensions, you might be tempted to add more features to get a better model, but more features can actually hurt your model.
- Each feature holds information that may or may not help your model predict accurately
- As you add more and more features, you need to add more and more training examples along the range of values for those features.
- The amount of training data needed increases exponentially with each added feature.
- That means that the volume of training data increases exponentially, and we need to make sure that the training data covers the same regions of the feature space as the prediction requests that it will receive
- All of these can reduce the ability of your model to generalize. Well, along with this, the number of trainable variables in the model also increases. To demonstrate this, let's look at an example of a binary classification model for the Cleveland heart disease data set when a single additional feature is added.

Model #1 (missing a single feature)

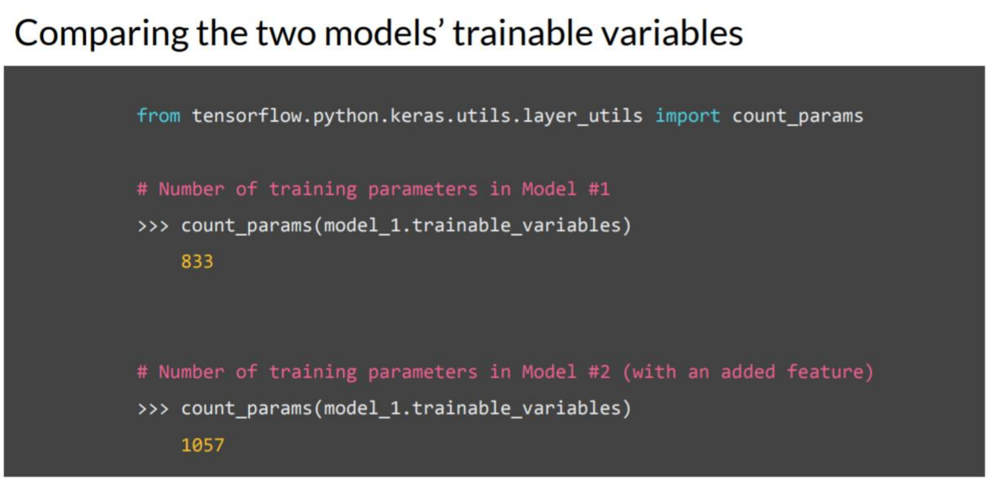

sex:InputLayer
category_encoding_6:CategoryEncoding
cp:InputLayer
category_encoding_1:CategoryEncoding
fbs:InputLayer
category_encoding_2:CategoryEncoding
restecg : InputLayer
category_encoding_3:CategoryEncoding
exang:InputLayer
category_encoding_4:CategoryEncoding
category_enconding_5:CategoryEncoding
ca:InputLayer
normalization:Normalization
age:InputLayer
normalization_1:Normalization
trestbps:InputLayer
normalization_2:Normalization
chol:InputLayer
normalization_3:Normalization
thalach:InputLayer
normalization_4:Normalization
oldpeak:InputLayer
slope:InputLayer
normalization_5:Normalization
concatenate: Concatenate
dense: Dense
dropout: dropout
dense_1: Dense

- Let's start by creating a structured classification model for the Cleveland heart disease data set. This dataset has 14 features to predict whether or not a patient has heart disease. This is a binary classification problem.
- This first model omits one of the original features called *thal.*

Model #2 (adds a new feature)


A new string categorical feature is added!
thal:InputLayer
string_lookup:StringLookup
sex:InputLayer
category_encoding_6:CategoryEncoding
cp:InputLayer
category_encoding_1:CategoryEncoding
fbs:InputLayer
category_encoding_2:CategoryEncoding
restecg : InputLayer
category_encoding_3:CategoryEncoding
exang:InputLayer
category_encoding_4:CategoryEncoding
category_enconding_5:CategoryEncoding
ca:InputLayer
normalization:Normalization
age:InputLayer
normalization_1:Normalization
trestbps:InputLayer
normalization_2:Normalization
chol:InputLayer
normalization_3:Normalization
thalach:InputLayer
normalization_4:Normalization
oldpeak:InputLayer
slope:InputLayer
normalization_5:Normalization
concatenate: Concatenate
dense: Dense
dropout: dropout
dense_1: Dense

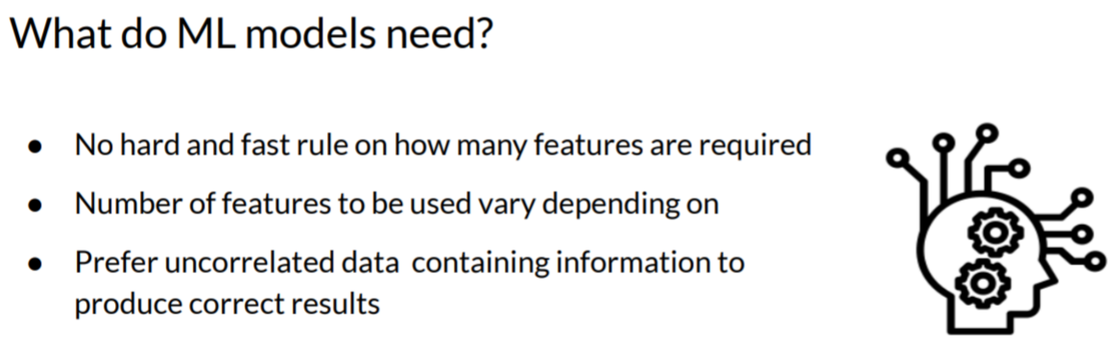
- Let's add back that single additional feature that we removed in the first model.
- For this, let's encode this as a categorical string feature using Keras preprocessing layers, which is available in TensorFlow.
- Next, let's see how adding it impacts your original model in terms of the number of trainable parameters.



Comparing the two models' trainable variables

```
from tensorflow.python.keras.utils.layer_utils import count_params

# Number of training parameters in Model #1
>>> count_params(model_1.trainable_variables)
    833


# Number of training parameters in Model #2 (with an added feature)
>>> count_params(model_1.trainable_variables)
    1057
```

- If you compare the number of trainable parameters between the two models, even adding only one feature results in a 27% increase.
- That's a considerable growth in the number of parameters to say the least. That will make training slower and more expensive.
- You'll also need to increase the size of the training data set which will make the training even slower and more expensive.

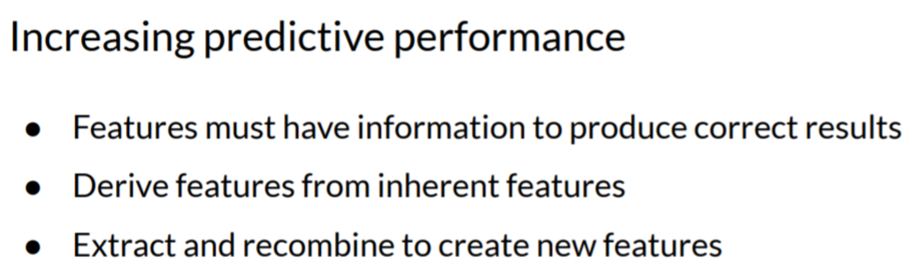


What do ML models need?

- No hard and fast rule on how many features are required
- Number of features to be used vary depending on
- Prefer uncorrelated data containing information to produce correct results

- The main point in this section is that when data dimensionality becomes too large, the performance of a classifier decreases and the demand for resources increases.
- The question, that is, what does too large really mean?
- Unfortunately, there is no fixed rule for how many features should be used in a machine learning problem.
- In fact, this depends on the amount of training data available, the variance in that data, the complexity of the decision surface and the type of classifier used.
- It also depends on which features actually contain predictive information that will help your model train.
- You want enough data with the best features and enough variety in the values of those features and enough predictive information in those features to maximize the performance of your model while simplifying it as much as possible.

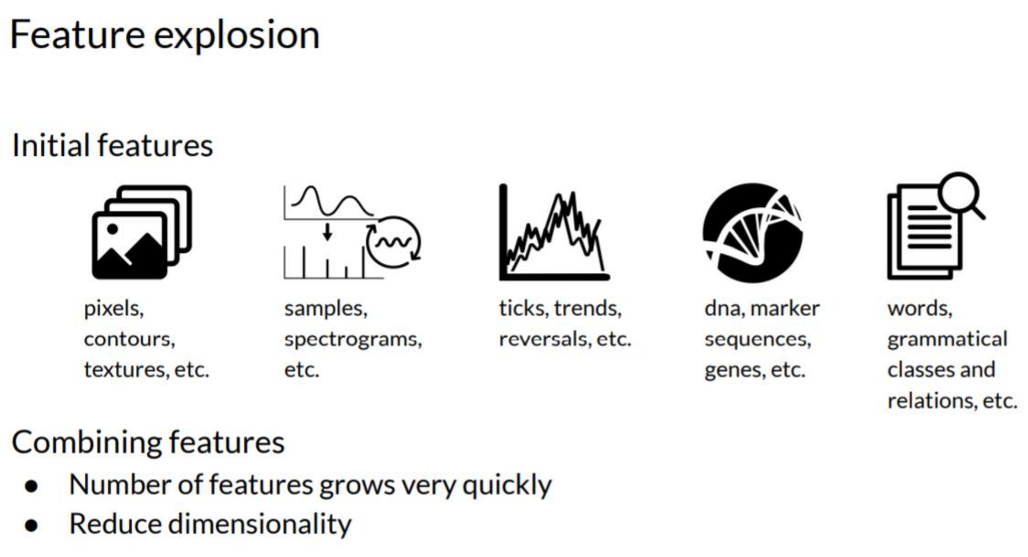
Manual Dimensionality Reduction



Increasing predictive performance

- Features must have information to produce correct results
- Derive features from inherent features
- Extract and recombine to create new features

- Now that we've discussed how your data has an impact on the performance of your models and the resources required to train and serve those models, let's look at some manual techniques for doing dimensionality reduction.
- When pre-processing a set of features to create a new feature set, it's important to retain as much predictive information as possible
- Without predictive information, all the data in the world won't help your model learn.
- Features must be representative of the predictive information in the data set. This information also needs to be in a form that will help your model learn.
- While some inherent features can be obtained directly from raw data, you often need derived features, normalized, engineered or embedded features.
- **A poor model fed with important features will perform better than a fantastic model fed with low quality or bad features.**



Feature explosion

Initial features

pixels, contours, textures, etc.

samples, spectrograms, etc.

ticks, trends, reversals, etc.

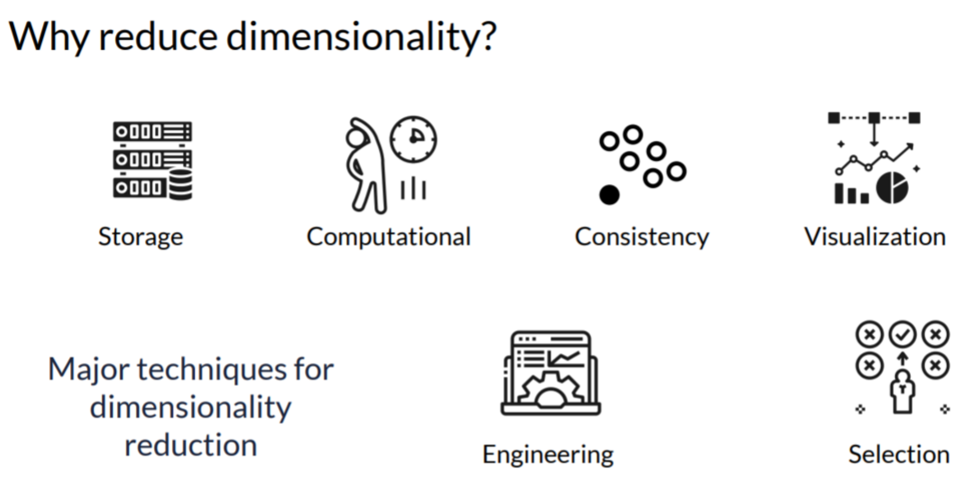
dna, marker sequences, genes, etc.

words, grammatical classes and relations, etc.

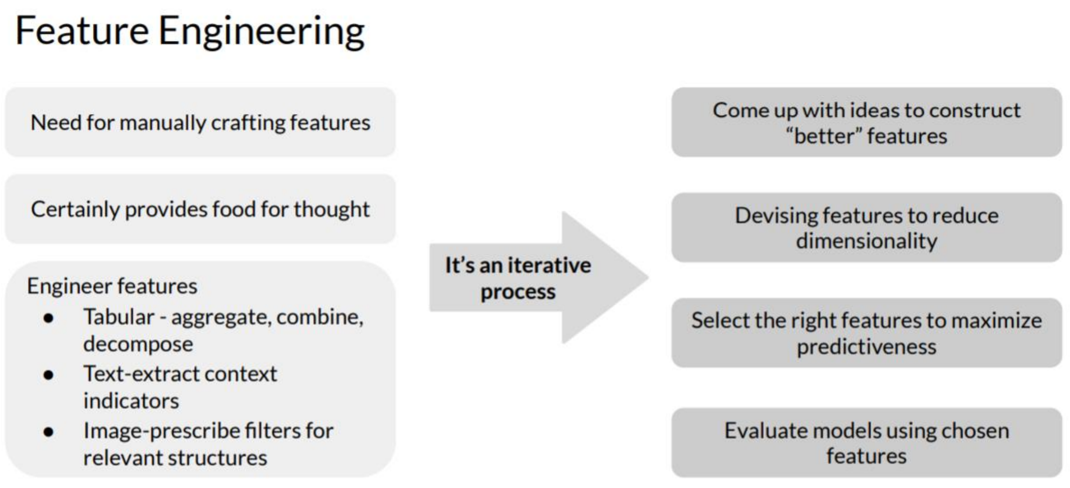
Combining features

- Number of features grows very quickly
- Reduce dimensionality

- Many domains involve huge numbers of features and dimensions.
- Often the first pick of features is an expression of domain knowledge, which can often result in more features than we really need or want.
- As we saw in our discussion of the curse of dimensionality this has inherent drawbacks. This means that you need to reduce dimensionality, or more precisely, the number of features you're including in your dataset while retaining or improving the amount of predictive information contained in the data.


Why reduce dimensionality?
Storage
Computational
Consistency
Visualization
Major techniques for dimensionality reduction
Engineering
Selection

- Dimensionality reduction looks for patterns and data and uses these patterns to re-express the data in a lower dimensional form.
- This makes computation much more efficient, which can be a significant factor in a world of big models and big data sets.
- However, dimensionality reduction's most essential function is to reduce the data set to its bare bones, discarding noisy features that cause significant problems for supervised learning tasks like regression and classification.
- In many real world applications it is the dimensionality reduction that makes predictions possible.
- Your data collection and management infrastructure will be simplified, also. Another factor to consider is that some algorithms do not perform well when we have large dimensions.
- It also reduces multicollinearity by removing redundant features.
- It helps when we're trying to visualize the data. And as we discussed earlier, it isn't easy to visualize data in higher dimensions, so reducing our space to 2D or 3D may help us to plot and observe patterns more clearly.
- Feature engineering helps meet these requirements. It builds valuable information from raw data by **reformatting, combining and transforming primary features into new ones** until it yields a new set of data that results in a better model.
- In addition, features selection examines a set of potential features, select some of them and discards the rest. Feature selection is applied either to prevent redundancy and or to remove irrelevancy in the original features, or just get to a limited number of features to avoid issues.
- Since we've already seen various feature selection techniques, let's look instead at feature engineering.



Feature Engineering

Need for manually crafting features

Certainly provides food for thought

Engineer features
- Tabular - aggregate, combine, decompose
- Text-extract context indicators
- Image-prescribe filters for relevant structures

It's an iterative process

Come up with ideas to construct "better" features

Devising features to reduce dimensionality

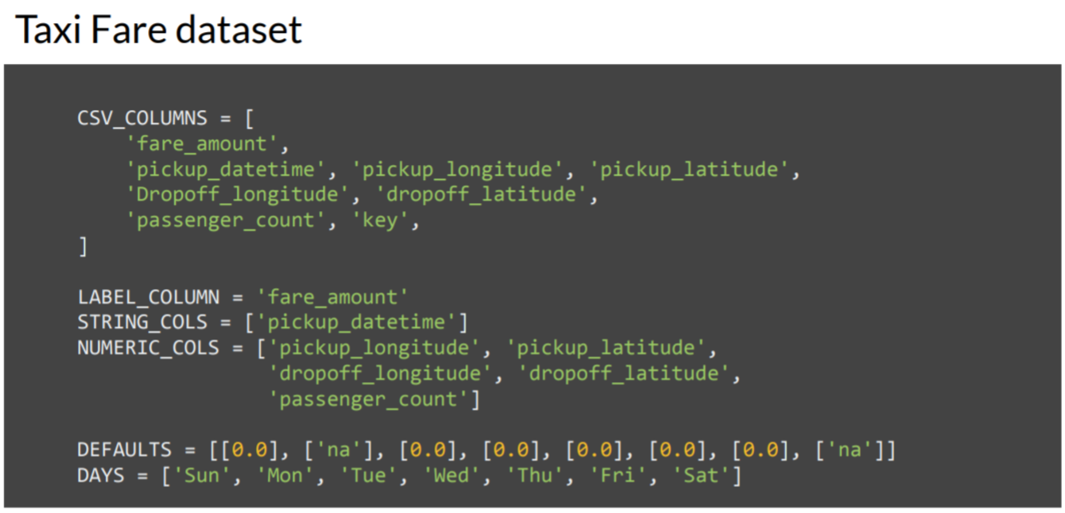
Select the right features to maximize predictiveness

Evaluate models using chosen features

- The best results come down to you, the practitioner crafting the features. This is one of the areas where machine learning engineering is somewhat of an **art** form.

- Feature importance and feature selection can help inform you about the objective utility of features, but those features have to come from somewhere.
- You often need to manually create them. This requires spending a lot of time with the actual sample data and thinking about the underlying form of the problem, the structures in the data and how best to express them for predictive modeling algorithms.
- With tabular data, it often means a mixture of aggregating and or combining features to create new features, and decomposing or splitting features to also create new features.
- With textual data, it often means devising document or content specific indicators relevant to the problem.
- With image data, it can often mean using filters to pick out relevant structures like pixels, contours, shapes and textures.
- It tends to be an iterative process that involves data selection and model evaluation again and again.
- The process usually **starts with brainstorming** features. Here you really get into the problem, look at a lot of data, study feature engineering on other problems and see what you can learn.
- Then you move on to devising new features. It depends on your problem, but you may use automatic feature extraction, manual feature engineering or a mixture of the two.
- Next you pick the right features using feature important scoring and feature selection methods to prepare one or more views of your data.
- Finally, you measure the model's performance on unseen data using the chosen features.

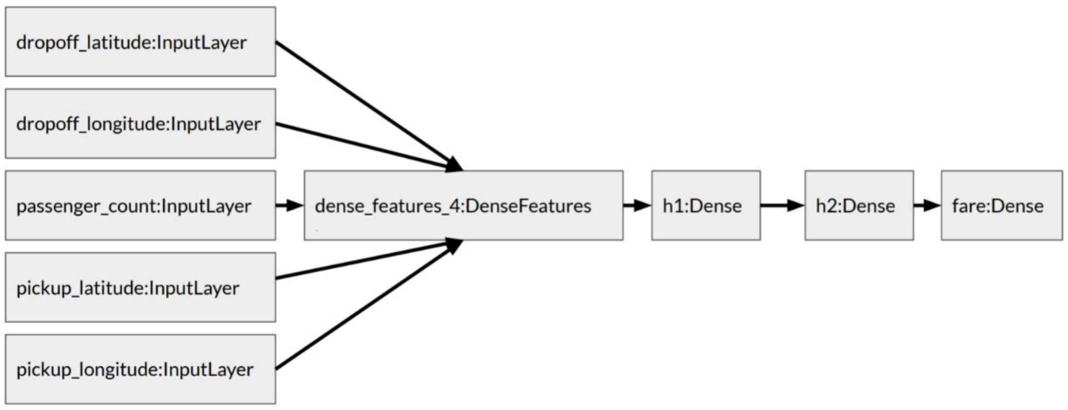
Manual Dimensionality Reduction: Case Study


Taxi Fare dataset

```
CSV_COLUMNS = [
    'fare_amount',
    'pickup_datetime', 'pickup_longitude', 'pickup_latitude',
    'Dropoff_longitude', 'dropoff_latitude',
    'passenger_count', 'key',
]

LABEL_COLUMN = 'fare_amount'
STRING_COLS = ['pickup_datetime']
NUMERIC_COLS = ['pickup_longitude', 'pickup_latitude',
                'dropoff_longitude', 'dropoff_latitude',
                'passenger_count']

DEFAULTS = [[0.0], ['na'], [0.0], [0.0], [0.0], [0.0], [0.0], ['na']]
DAYS = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

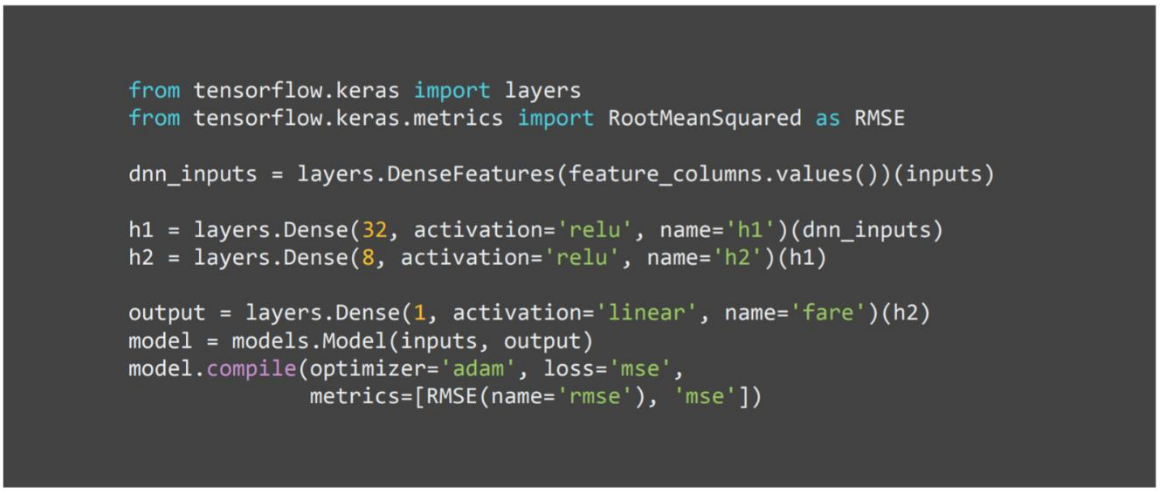
- Now you've seen that the dimensionality of your data has an impact on the performance of your models and the resources required to train and serve those models.
- This is even more important when dealing with resource constraints scenarios such as mobile deployments.
- So you need to carefully manage the dimensionality of your data, which often means looking for ways to reduce it.
- Let's look now at some manual techniques for doing dimensionality reduction. Let's look at a concrete example using the taxi fare data set.
- The data set consists of 106,545 taxi rides. The objective is to predict the fares of each ride based on a variety of features such as time and location of pickup, time travel and distance, number of passengers etc.
- As usual, the first steps are downloading the data set which is in CSV format separating the variables into string and numeric types, and defining useful constants and parameters.

Build the model in Keras


dropoff_latitude:InputLayer
dropoff_longitude:InputLayer
passenger_count:InputLayer
pickup_latitude:InputLayer
pickup_longitude:InputLayer
dense_features_4:DenseFeatures
h1:Dense
h2:Dense
fare:Dense

- Let's build a baseline model to predict the fare. We'll try using these features: Drop off, latitude, drop off longitude passenger count, pick up latitude and pick up longitude.
- The network consists of a concatenation of dense hidden layers with the last one producing a fare prediction output.

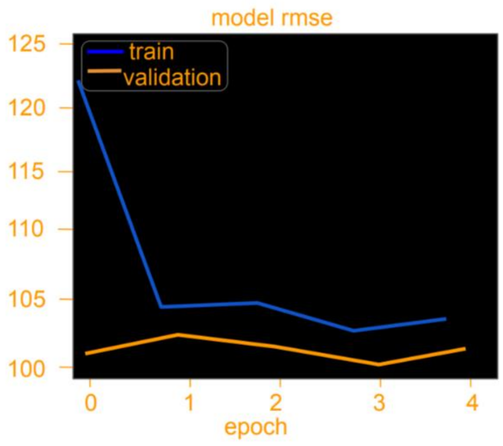
Build a baseline model using raw features



```
from tensorflow.keras import layers
from tensorflow.keras.metrics import RootMeanSquared as RMSE

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

h1 = layers.Dense(32, activation='relu', name='h1')(dnn_inputs)
h2 = layers.Dense(8, activation='relu', name='h2')(h1)

output = layers.Dense(1, activation='linear', name='fare')(h2)
model = models.Model(inputs, output)
model.compile(optimizer='adam', loss='mse',
              metrics=[RMSE(name='rmse'), 'mse'])
```

- We'll build the model in Keras using the **functional** API
- Unlike the sequential API, you'll need to specify the input and hidden layers.
- Note that you're creating a linear regression baseline model with no feature engineering
- As a quick reminder, a baseline model is a naïve implementation that helps with setting expectations for model performance.

Train the model



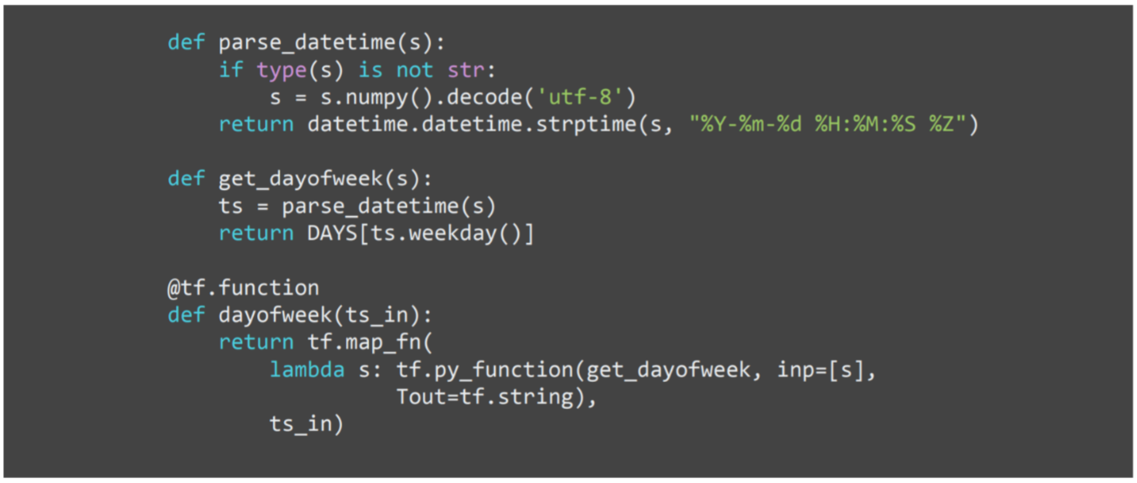
- After setting up the model for training and creating the data sets, you're ready to train the baseline model. To train the model simply call model.fit.
- Let's look at training and validation performance using the root mean squared error loss over training epochs. Ideally you want the validation RMSE to be close to the training set.

Increasing model performance with Feature Engineering

- Carefully craft features for the data types
  - Temporal (pickup date & time)
  - Geographical (latitude and longitude)

- Now, let's try to improve the model. To improve the model's performance, let's create two new features, engineering types, temporal and geographical

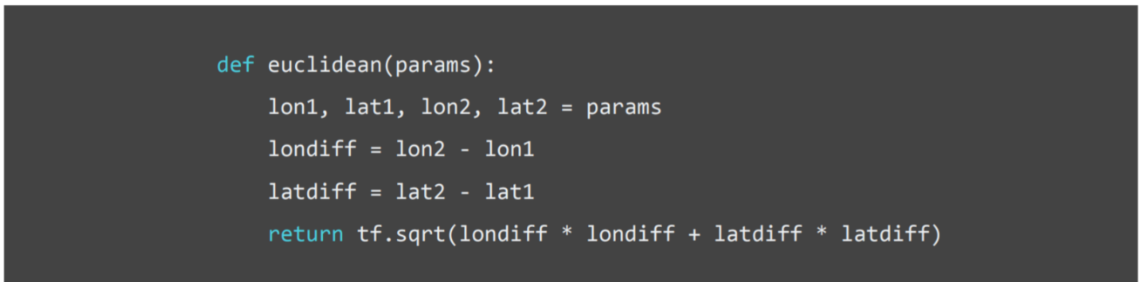
Handling temporal features



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

def get_dayofweek(s):
    ts = parse_datetime(s)
    return DAYS[ts.weekday()]

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

- For example, we will work with the temporal feature, pick up date time as a string and we will need to handle this within the model.
- First, you'll include pickup date time as a feature and then you'll need to modify the model to handle it as a string feature.

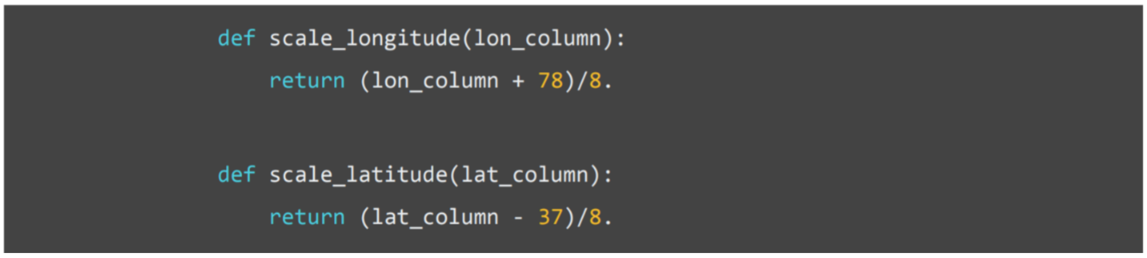
Geolocational features



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

- The pickup or drop off longitude and latitude data are crucial to predicting the fare amount since fare amounts in New York City taxis are largely determined by the distance traveled.
- As such we need to teach the model of the Euclidean distance between the pickup and drop off points.
- Recall that latitude and longitude allows us to specify any location on Earth using a set of coordinates. The dataset contains information regarding the pickup and drop off coordinates.
- However, there is no information regarding the distance between the pickup and drop off points.
- So let's create a new feature that calculates the distance between each pair of pick up and drop off points.
- You can do this using the Euclidean distance, which is a straight line distance between any two coordinate points, but note that this will only be a rough indicator of the actual driving distance.

Scaling latitude and longitude



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


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

- It's very important for numerical variables to get scaled before they're fed into the neural network.
- Let's use min-max scaling, also called normalization, on the geolocation features.
- Later in our model, you'll see that these values are shifted and rescaled so they end up ranging from 0 to 1.
- We'll use domain knowledge to create a function named *scale_longitude* where you pass all the longitude values and add 78 to each value. Note that are scaling longitude values range from negative 72 to negative 78. So the value 78 is the maximum longitudinal value. The difference or delta between negative 70 and negative 78 is eight.
- The function adds 78 to each longitudinal value and then divides by eight to return a scaled value.
- Similarly, let's create a function named *scale_latitude* where you pass in all the latitudinal values and subtract 37 from each value. Note that are scaling latitude range is from 37 to 45.
- Thus the value 37 is the minimal latitudinal value. The delta or difference between negative 37 and negative 45 also happens to be eight.
- The function that subtracts 37 from each latitudinal value, and then divides by eight to return a scaled value.