Golden Path to Machine Learning

Created by Kristie Wirth

> *Last updated: 2020-10-21*
>
> *Have feedback? We'd love to hear it! Leave your comments here.*

Introduction

What is the Golden Path to Machine Learning?

This course is a combination of learning materials (external & internal) and programming exercises to get you familiar with the core concepts of machine learning. We've heard a lot of engineers express interest in machine learning, and we'd like to share our knowledge with you, so you too can build and understand basic models!

Why would you want to take this course?

- The need for folks skilled in machine learning is going to continue to increase over time.
- You can likely apply machine learning techniques to your current role to be more effective and create creative new projects.
- It's a fun area of work with unique and interesting problems to solve.

Prerequisites

- Basic knowledge of Python

How is the course structured?

The course is meant to be worked through sequentially, though feel free to skip any sections if you're already quite familiar with the material. Some sections will be more concept based, while others will focus on coding steps . Many sections will have extra links at the end if you'd like to do a deeper dive on a given topic.

When can you use machine learning?

There's two types of machine learning – supervised and unsupervised. You would use them in different situations.

Supervised learning

For when you want to make predictions.

Predictions is a broad word, but there are so many things you can do with supervised learning. You can predict numerical values, like the lifetime value of a customer. You can predict categorical values, like the country a user lives in (for those missing that data).

To make a supervised learning model, you will need:

1. A dataset with one row per entity you're making a prediction about (e.g. user, zap, support ticket).
2. Other information about that entity – this is how you will make the predictions (e.g. role, domain, browser).
3. The "correct answers" for your dataset (e.g. if you're trying to predict a user's country, first you'll need a dataset with other users and their countries for your model to learn from).

Unsupervised learning

For when you want to make meaningful categories.

Unlike supervised learning, in unsupervised learning you don't have the answers already. Instead, what you have is a dataset with some information, and you want to make categories out of it, such as grouping survey responses into themes or creating groupings of similar customers.

To make an unsupervised learning model, you will need #1 & #2 listed above, but unlike supervised models, you do not need #3.

Acquiring your data

Before you can do any fancy machine learning techniques, you'll need to be able to acquire the dataset for your model to learn patterns from.

We'll use this publicly available dataset from Kaggle.

Exploring your data

Some people say machine learning is a combination of science + art. The science is all the repetitive work, you'll most likely end up testing similar models and using similar methods of evaluation over time. But every time you build a model, the data is unique, and requires different treatment. This is where the art of it comes in - how you set up the data itself can be one of the most time consuming parts, and it can require some creativity to figure out the best ways to do so. We also think it's one of the most fun parts too!

Let's build a model that predicts whether or not a person survived the Titanic sinking. From now, the information we're using to predict this (age, fair) we will call **features**, and the information we're trying to predict (survival) we will call our **target**.

A note about target variables - Sometimes you won't have a clean target variable, or you'll want to predict something that you don't have data for. Let's say you want to predict the difficulty of a support ticket, but you don't have a dataset of tickets labeled with their corresponding difficulty. In this type of situation you have two options - you can either get a group of folks together to label past tickets with difficulty or, you can create your own target variable from other information.

If you get folks to hand label your data, make sure you first develop some guidelines on how to do that labeling, so that the labels on consistent. You may even want to have multiple folks classify the same ticket, so you can then average their votes later on.

If you want to create your own target variable, you have to be creative in thinking about what data you do have access to. For example, if you have access to the length of time a customer champion actively worked on a given support ticket, you could assume that more time = more difficult, and thus use time as a target variable that approximates difficulty!

Learn about your data

First off, let's import pandas - a Python package that manipulates and explores data. You'll end up using pandas quite a lot whenever you work with data. We'll also import datto - a Python package that Kristie actually created that will speed up a lot of the ML processes we'll cover here and later on.

Check out the source code for datto if you'd like to get into more detail on how these methods work.

```
import pandas as pd

import datto as dt

df = pd.read_csv('train.csv')
```

Let's see how big our dataset is:

```
df.shape
```

Take a look at the first few rows:

```
df.head(10)
```

Let's clean those column names for easier referencing later on.

```
ct = dt.CleanText()
df = ct.clean_column_names(df)
```

There, that's better. Now we don't have any spaces, everything is lowercase and uses underscores, and we've removed special characters.

Now, how many different values are there to predict?

```
df["survived"].unique()
```

Great, 2 values keeps our model nice & simple.

Let's make sure we indeed have one row per passenger:

```
df[df.duplicated(['passengerid'])].sort_values(by='passengerid')
```

Great, looks like we don't have any duplicate rows. If you do find duplicates, you'll want to use `drop_duplicates` by a subset of `passengerid`.

Now that you've gotten a better idea of what your data looks like, let's move on to cleaning up your data.

Cleaning your data

Prepare your data for modeling

Let's check the data types of each column to see if they're correct:

```
df.info()
```

Looks like `pclass` and `passengerid` should be categories. Let's fix these.

```
df = ct.fix_col_data_type(df, 'passengerid', 'str')
df = ct.fix_col_data_type(df, 'pclass', 'str')
```

If you recheck the data types, you'll see that we indeed changed this type to be numeric!

Let's see if we should exclude any columns from our analysis. Some reasons we may want to exclude values include too many unique values, low variance in numerical values, or a large percentage of nulls.

```
eda = dt.Eda()
eda.find_cols_to_exclude(df)
```

We definitely won't want `passengerid` as a feature, so we can drop those now. `ticket` has so many unique values we probably can't use as it is. `cabin` has many nulls so let's drop that too. We'll keep `name` for later language processing.

```
df.drop('passengerid', inplace=True, axis=1)
df.drop('ticket', inplace=True, axis=1)
df.drop('cabin', inplace=True, axis=1)
```

One thing that will cause an error when building a model is having null values. Let's check for counts of nulls and fill those values with a placeholder.

Checking for nulls in each column:

```
for col in df.columns:
    num_nulls = df[df[col].isnull()].shape[0]
    if num_nulls > 0:
        print(col)
        print(num_nulls)
        print('-------------')
```

Okay, so we'll need to fill null values for a few different columns. Note that you never want to delete null values! There could be something different about users with null values, and deleting them could have you lose valuable information from your model.

For the numerical values, what we want to fill the nulls with will depend on what the variable is. Let's fill age with a mean, and embarked with a placeholder of 'unknown'.

```
df["age"].fillna(
    df["age"].mean(), inplace=True
)
df["embarked"].fillna('unknown', inplace=True)
```

Let's check if we have any correlated features. Correlated features can mess up a model sometimes.

```
eda.find_correlated_features(df)
```

Looks like the correlation between our numeric features is relatively low, so nothing to do here! Let's move on to some general discussion of types of machine learning models you may use.

Optional additional resource: this tutorial of more data cleaning techniques

---

Supervised learning models

Let's step back for a moment and talk briefly about how machine learning models work under the hood. We'll circle back to building your model, so save your code from above!

While you can program a machine learning model without knowing any of the theory, the more you know about how these work, the better you can be at understanding which model to use and why a certain model may be performing how it is.

So, a machine learning model is the method you use to get your predictions. A supervised machine learning model will learn patterns in your data that best predict some given information.

Let's compare this to making a set of heuristics – something you may already do in your work. For example, based on what we know about our passengers, we might make a set of heuristics that look something like this:

```
if age > 60:
    survived = False
elif pclass = 3:
    survived = True
else:
    survived = False
```

(These rules are completely made up as an example, don't assume they're true!)

A machine learning model is an alternative to making a set of heuristics like this. Instead of creating the heuristics ourselves, the model learns these heuristics automatically from our data and figures out the best set of rules and combinations of features that it will use to make predictions. Let's learn very broadly about a few types of ML models, keeping in mind that you could take a whole course learning about the intricacies of each of these!

There are two types of each of the following models - **classifiers** and **regressors.** If you are trying to predict a categorical target, you will use a classifier, and if you are trying to predict a numerical value, you will use a regressor.

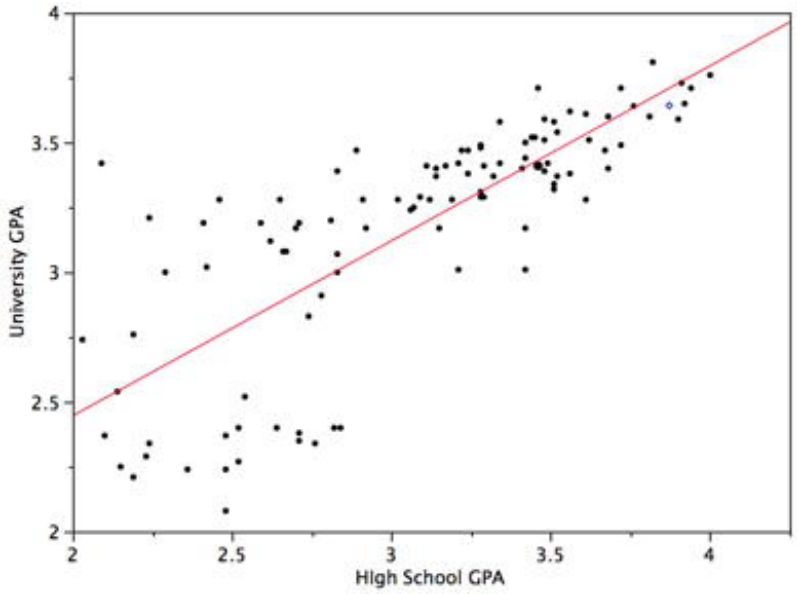
Optional additional resource: listing of all the supervised learning models that sklearn (Python package) has available

Regressions

The classifier version of a regression is called logistic regression, while the regressor version is called linear regression.

In a both type of regressions, we create something called a line of best fit which is then used to predict the target.

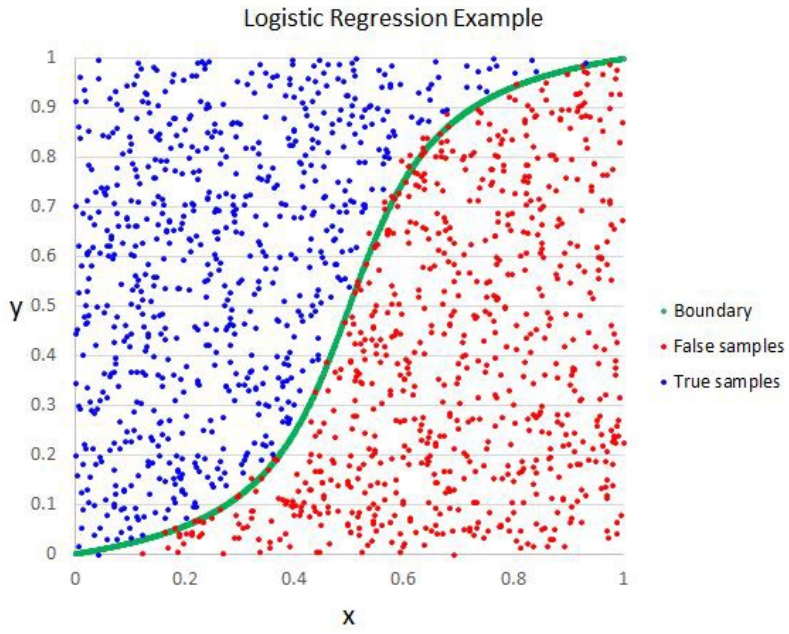
For a linear regression, you can imagine plotting all your data on a graph, and then trying to draw the best line that goes through as many data points as possible.



(Source)

While linear regression predicts a numerical value for the target, a logistic regression predicts a probability of each category. It also creates a line on your graph, but this line looks different. The line attempts to separate categories as best as possible. Everything to the right/underneath the line is predicted as a higher probability of being True, and everything to the left/above the line is predicted as a higher probability of being False.

For example, in our work above, we would predict the probability of someone surviving, and use that probability to ultimately decide whether it's a higher probability they are a survivor or a higher probability they are not a survivor.



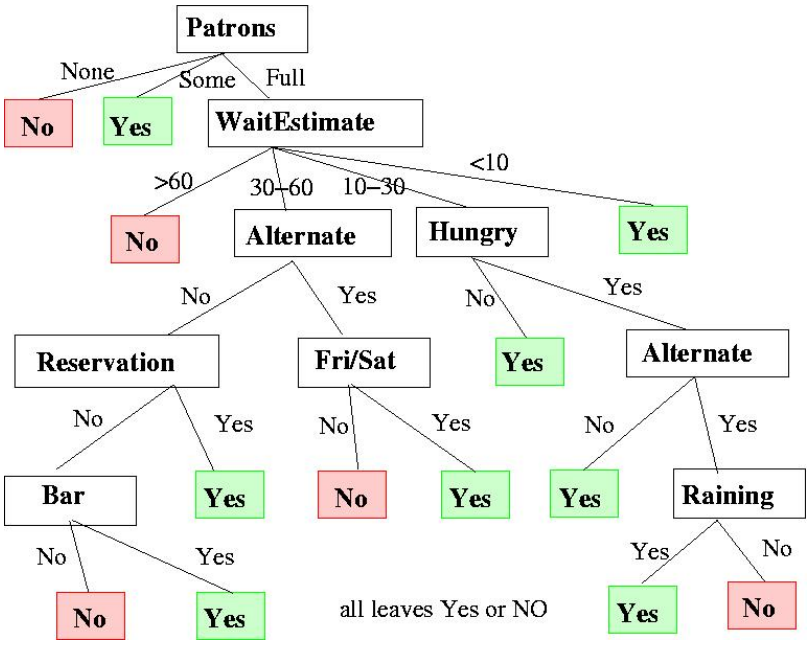
(Source)

Optional additional resources:

- Other types of regressions
- Interpretting regression coefficients
- Reading a logistic regression graph

Decision trees

You can think of a decision tree as a sort of flow chart, with various decision making pieces at each step. The features are used as the questions that guide you down the tree, where you ultimately end up with a value or probability for your target variable at the bottom. Each split in the tree is based on the answer to a single question, i.e. a single feature. The splits are chosen automatically by the model learning which types of splits best separate the target variable's values.



(Source)

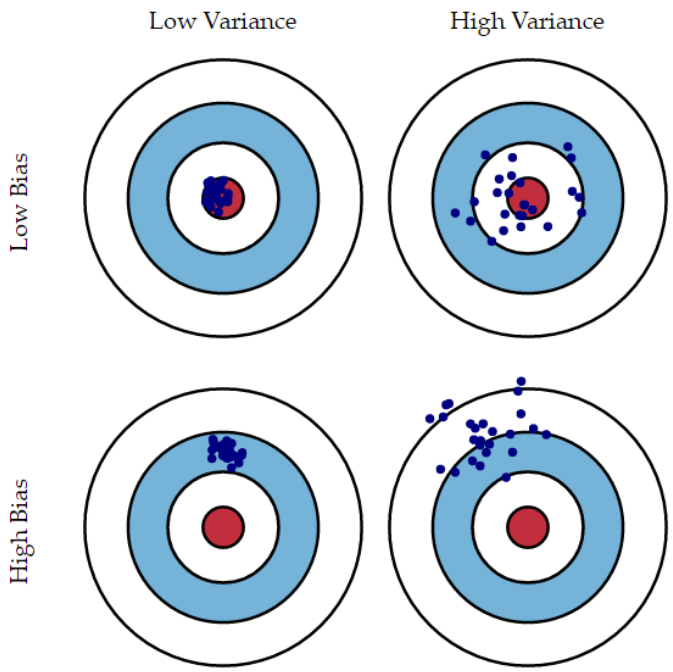
Optional additional resource: Kristie's lecture on decision trees

A quick aside about bias & variance...

To understand the following few models it's helpful to take a moment to explain two terms - bias & variance.

Bias is how far off your predictions are the true value. **Variance** is how different your predictions are from one another.

In other words, bias how far you are from your target (the bulls eye), while variance is how spread out your darts are.



(Source)

Let's analyze the image above – the top row is low bias, as you can see, both have darts that are relatively close to the bulls eye (the correct answers for the target variable).

On the right hand column, we have darts that are quite spread out, or in other words, have high variance. And so on with the other rows & columns.

Models with high variance typically are paying too much attention to the data your model is learning from and don't generalize well to new incoming data. This is called **overfitting**.

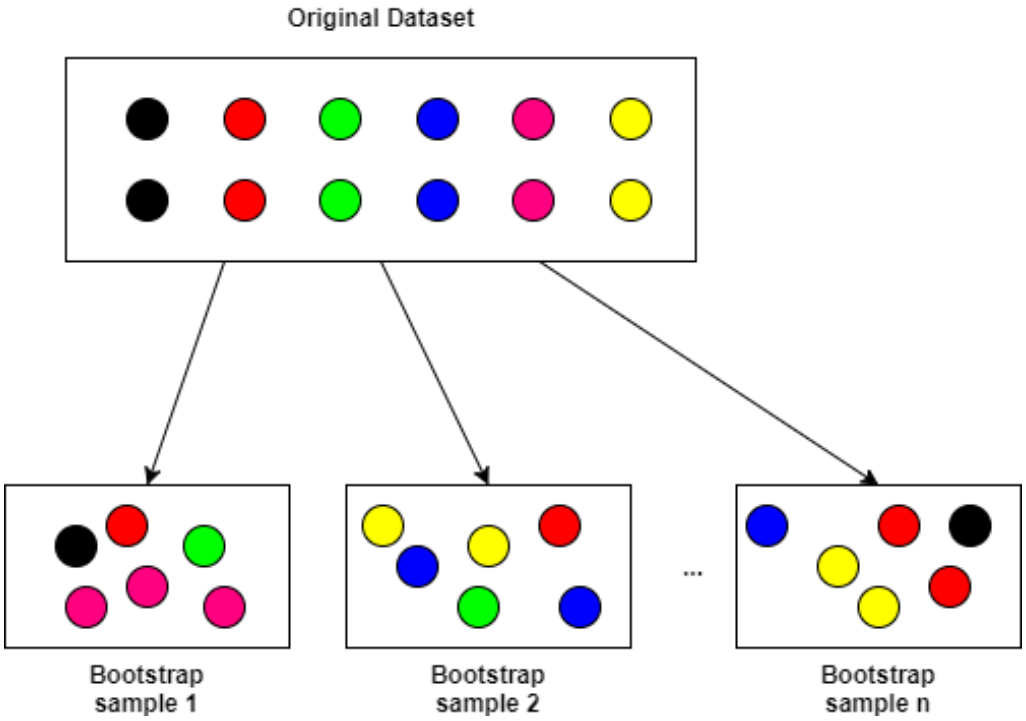
Models with high bias have the opposite problem - they aren't learning enough to capture the patterns in the data and thus have a good rate of correct predictions. This is called **underfitting.**

So this is a tradeoff, ideally you want to balance both bias & variance, and different models will have different advantages/disadvantages in these. Okay, let's move on to some more models...

Bagging

Bagging t is when you make a whole bunch of decision trees, and then take the average of those trees to make an average set of rules to use instead.

To make these different trees, you resample your data using **bootstrapping** and then make a tree from each new bootstrapped sample. This means that you pull a sample from your data, and then replace it, so you end up with a new dataset of the same size, but with various samples repeated. It's like picking a random ball from a bag, and then each time putting the ball back in the bag to potentially pick those same balls again.



(Source)

In the image above, you would create a decision tree from each bootstrapped sample, and then later average those trees together. As you might imagine, bagging (and the following more complex tree based methods) typically perform better than a single decision tree as they help average out any extremes and get the most useful splits from multiple runs.

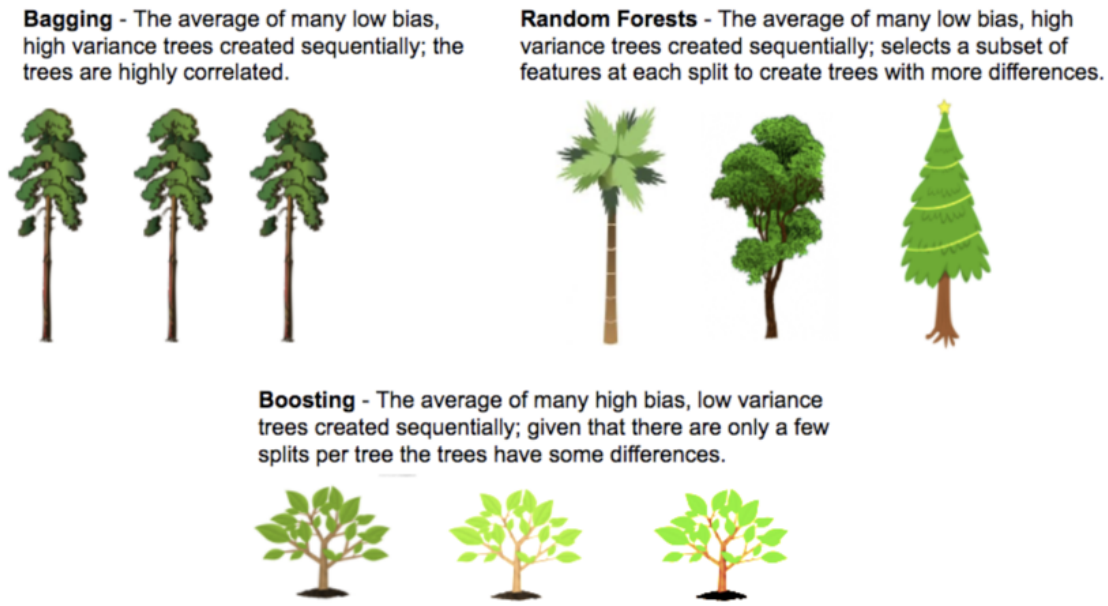
Random forests

Random forests work very similarly to boosting with one key difference – instead of using every single feature in your dataset to build each new tree, the model randomly selects only a few of the features available, and builds a model using that subset of features. This leads to more diverse trees which tends to perform better than simple bagging.

Boosting

Boosting is another tree based method that is slightly different – instead of bootstrapping your data, each new decision tree is created based on the one before it. So after the first tree is built, the next tree is created by either making a new tree using the information the first tree learned (e.g. what splits are good). In random forests and bagging, each tree is pretty deep, meaning there are a lot of questions to go through in your flow chart before you reach an answer. In boosting, the trees tend to be short, meaning you may answer one or two questions and then arrive at your answer.

Here's an image to help illustrate the differences between random forests, bagging, and boosting:



(Created by Kristie)

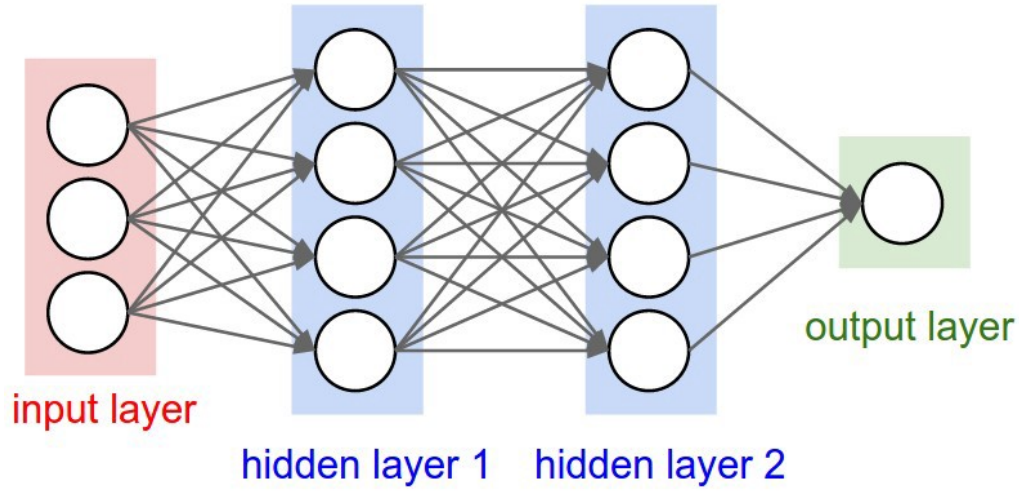
Neural networks

Alrighty, one last supervised learning model for you. It's the one I bet you've heard discussed, neural nets!

Neural networks are great for image data, video data, sound data, etc. E.g. if you want to classify an image as either a dog or a cat. They're also good for generating bodies of text. However, for most simple regression or classification problems, a neural network is most likely overkill. They're computationally expensive and often random forests or boosting will give great results with far less complexity, so we won't get too deep into going into how to code a neural net work in this course. However, if you'd like to go through a tutorial, this one is a great way to get started using a package called Keras.

There's several types of neural networks, but here's a broad overview of how a basic one works. First, you have an input layer - this is all your features in the data. You also have an output layer, this is the target you are trying to predict.

In the middle you have some number of hidden layers (how many layers to choose is mostly experimentation). To start, your model guesses some random weights to use in the hidden layer(s). The input layer is then multiplied by the hidden layer(s) and the output of that is compared to the real values of the target value. Then, we use something called **back propagation** to update the weights. This means that we calculate how far off our results were from the true results, and we use a particular calculation to adjust the weights in the hidden layer so that hopefully we can do a better job with our predictions next time. The number of times you go back and adjust those weights is called the number of **epochs**, and is also chosen mostly by experimenting with your data).



(Source)

Additional optional resources:

- Video overview of neural networks
- Video on CNNs (neural networks that focus on image classification)
- Video on RNNs (neural networks that focus on text generation)

Unsupervised learning models

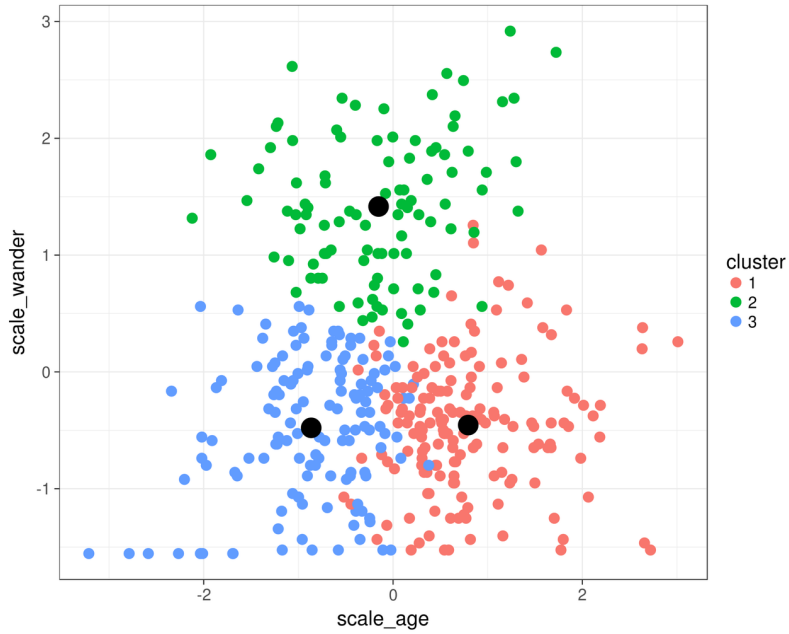
Refresher – Unsupervised learning models attempt to find distinct categories within your dataset, using the features available. There's no target variable, instead this model is simply creating categories. The idea is that the data points in a given category should be as similar as possible to each other, while being as different as possible from data points in other categories.

Optional additional resource: listing of all clustering models sklearn has available

K-means clustering

To make the separate categories in your data, this algorithm picks some number of clusters to create. It then picks that number of random data points. I.e. if you want 3 clusters, it chooses 3 random data points. These data points chosen are called **centroids**. Then, for each individual data point in your dataset (e.g. each user) it uses an equation to calculate the similarity between that data point and each of the centroids. This similarity is calculated by incorporating the features you have about each data point, such as features we added to our example dataset earlier, like `pclass` and `age`.

For each data point, it is assigned to the group where it is closest to that centroid. And so on, you assign each data point to a different centroid, thus making your categories. In the image below, the black dots are the centroids, and each data point would represent an item, user, etc. You can see that each data point has been assigned to a different color category, based on which centroid it is closest to.



(Source)

Of course, picking random points for your centroids will probably not result in great clusters, so part of the process of this model is choosing the number of times you want to run it, and then each time, the model adjusts and learns from itself, based on calculating how much the data points vary. With the end goal being to make as distinct categories as possible by having groups with all data points being quite similar to each other.

Optional additional resource: K-Means article

Non-negative matrix factorization (NMF)

NMF works by using linear algebra, so we won't get too deep into the details, as that is beyond the scope of this course. NMF specifically works great for clustering text data, though it can be used for other things.

Overall, the idea is that we have features representing (adjusted) counts of keywords in some text. What NMF does is it takes all those features we've gotten from the text and reduces them to only a few categories. It does this in a way that's somewhat similar to how neural networks work, by multiplying some matrices together, but then going back and determining the best weights to multiple by that create distinct categories.

Optional additional resource: LDA/ NMF article

Setting up for your model

Splitting the dataset

Alrighty, now that we've taken some time to go over concepts, let's dive back into the code!

X will be our feature set, and y will be our target. Let's split our datasets into those separate chunks:

```
y = df["survived"]
X = df.drop("survived", axis=1)
```

In order to evaluate how well your model is doing, you'll need a train set and a test set of data. This is done by splitting your data into two different datasets, with the train set being larger than the test set.

The reason for this split is that you want your model to learn patterns from one set of data, but then, in order to mimic how it might perform out in the real world, you want to test it on some data it has never seen before. And you want to test it on some data you have the "correct" answers for, so that you can evaluate approximately how well it might do if you were to deploy it.

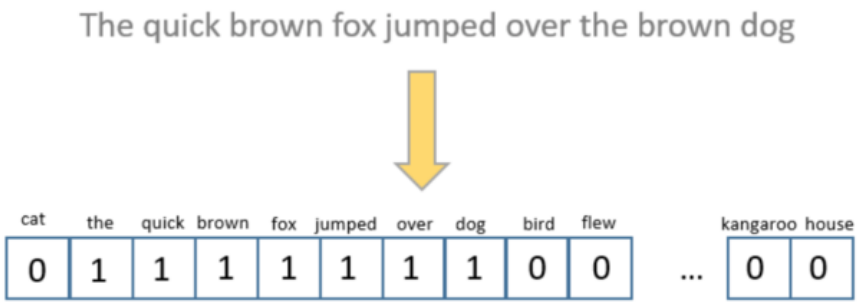
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

An important idea to remember is that you want to keep these datasets entirely separate until the very end, when you test (once!) how well your model is doing.

Natural language processing

`name` isn't really useful as a categorical feature - it has many different values by each person. However, we can comb the text to see if there are keywords in how a name is formatted that are useful in predicting whether someone survived.

To do that, we need to transform this one column into many columns, that represent adjusted counts of each word in the `name`.



(Source)

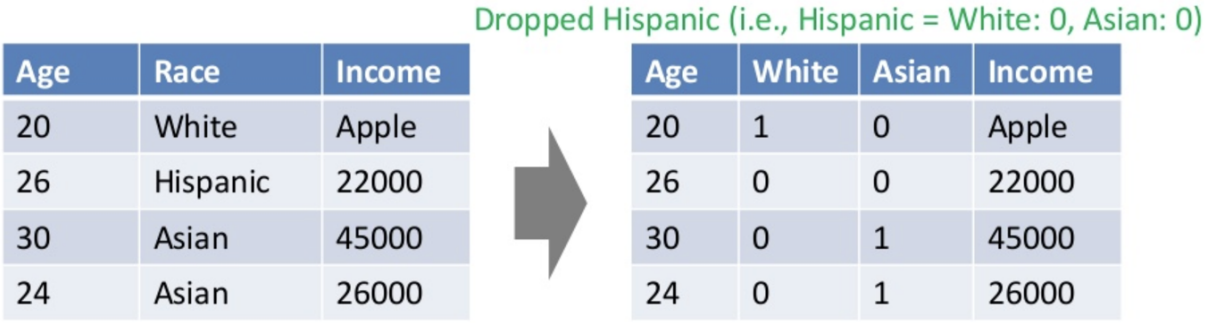
Let's transform our text (`name`) into separate columns of adjusted word counts instead. As part of this process, we will **lemmatize** the text – this means changing everything into its root word to standardize the word counts (e.g. `running` to `run`, `leapt` to `leap`).

```
from sklearn.feature_extraction.text import TfidfVectorizer

# The vectorizer that will transform text into columns of adjusted word frequencies
vectorizer = TfidfVectorizer(
    tokenizer=ct.lematize,
    # Means you can have individual words or phrases of 2-3 words
    ngram_range=(1, 3),
    # Means each word must appear in at least 5 different data points
```

```
    min_df=5,
    # Means each word cannot appear in more than 40% of data points
    max_df=0.4
)

# Fit + transform = learn from train data, then create transformations
train_vectors = vectorizer.fit_transform(X_train["name"]).todense()
# Transform only - you don't want to learn anything from the test data
test_vectors = vectorizer.transform(X_test["name"]).todense()

# Let's get the words used as features
words = vectorizer.get_feature_names()

# Then we can morph the new columns back into a dataframe,
# and label the columns with the words used
X_train_vectors = pd.DataFrame(train_vectors, columns=words, index=X_train.index)
X_test_vectors = pd.DataFrame(test_vectors, columns=words, index=X_test.index)

# And now we can drop the original column
X_train.drop("name", inplace=True, axis=1)
X_test.drop("name", inplace=True, axis=1)
```

Handling categorical features

The model also can't handle categorical features as is – everything needs to be numbers! So what we can do is create **dummy variables**. This is when you transform one column into several columns with 1s and 0s that represent that initial column. Note that it's redundant to have every variable be a column so you drop one, so you drop one. Then that dropped feature is represented by all 0s.



(Source)

```
# Make dummies using the train set
X_train_dummies = pd.get_dummies(X_train, drop_first=False)
# And again, you never want to learn from the test set,
# so you reindex to match the train set
X_test_dummies = pd.get_dummies(X_test, drop_first=False).reindex(
    columns=X_train_dummies.columns, fill_value=0
)
```

Note that running dummies on the whole dataset as done above will make dummies for any categorical variables and keep any numerical values as is. After you make these columns, now you can combine the the word vector columns

and the dummy columns into one final train set and one final test set.

```
X_train_combined = pd.concat([X_train_vectors, X_train_dummies], axis=1)
X_test_combined = pd.concat([X_test_vectors, X_test_dummies], axis=1)
```

Training your model

Now that you've prepared your data, the actual part of training a simple model on the data is quite simple!

You need to instantiate a model (such as a Logistic Regression):

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
```

Then train that model:

```
lr.fit(X_train_combined, y_train)
```

Now give it the test set to make predictions on!

```
y_predicted = lr.predict(X_test_combined)
```

You can also score how well that model is doing. There are a lot of different scoring methods you can use, but let's use precision & recall.

Evaluating your model

Classifiers

Here's a few common scoring metrics for classifiers:

- **Precision** – out of everything you predicted as true, what percent were actually true?
- **Recall** – out of everything that was true (e.g. people who are survivors), what percent did you "catch" / predict as true?
- **Accuracy** – out of all the predictions you made, what percent were correct?

We'll walk through some examples to better understand these. Let's say you have a dataset of 100 folks.

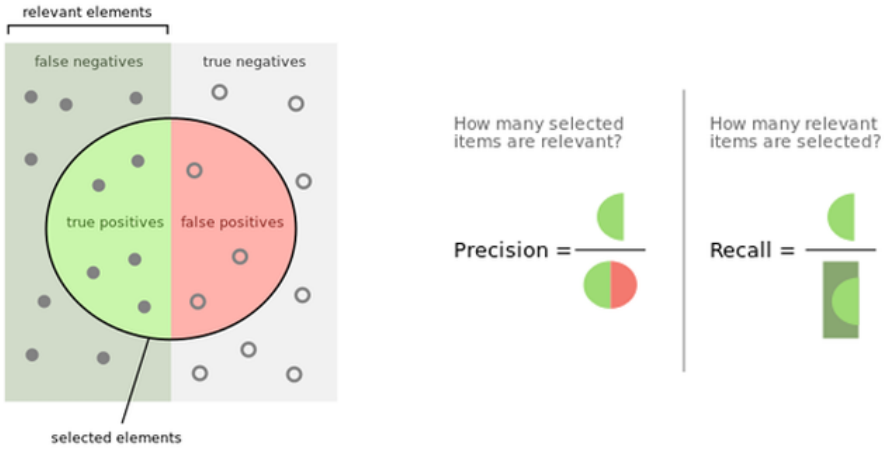
40 of them survived. 60 did not not.

You predict that 20 of them are survivors. Of those 20 you predicted as survivors, only 13 were actually survivors.

So you would have a precision of 13/20, or 65%.

You correctly predicted 13 were survivors, out of 40 total survivors in the dataset. So you have a recall of 13/40, or 33%.

Doing some math to figure out the other pieces, this means that you correctly predicted 13 folks were survivors, and you were correct in predicting 53 folks were not survivors. This means your accuracy was (13+53)/100, or 67%.



(Source)
Using our example numbers, in the above diagram, the light green circle half is 13, the red circle half is 7, the dark green side is 27, and the light gray side is 53.

Choosing classification metrics

Accuracy is the most well known metric here, and, it can be a problematic metric. For this reason I recommend you either avoid it entirely, or only use it in combination with the other metrics listed to get a fuller picture of how your model is doing.

Here's one example where accuracy is misleading – if we continue with the above example, let's say your model predicts that everyone is not a survivor. This would give you an accuracy of 60%! (60 are not survivors, so you are correct 60 times). Which sounds relatively decent, however, a model that only predicts False is not particularly useful. Checking precision and recall here would alert you to this fact.

In terms of deciding between precision and recall, you'll want to think about your use case and which one is more important. Let's say you're trying to predict email spam. Here, it's much worse to classify something as spam when it isn't spam, because you would miss potentially important emails. So, precision is more important – you want to make sure you are as spot on in your predictions as possible.

As another example, let's say you're trying to predict whether someone has a highly contagious illness. In this case, missing a patient who is sick is much worse than accidentally telling someone they are sick when they aren't. In this case, recall is a higher priority. You want to catch as many of the sick patients as possible with your model.

Most likely you will want to measure both of these metrics to balance them somewhat, but knowing your use case will help you decide what to optimize your model for.

Optional additional resource: article on classification metrics

Regressors

Root mean squared error (RMSE)

First off, this is a measurement of error, so it's important to note that the lower this score is, the better. Here are the steps to calculating it to illustrate what it means (though you won't have to do this manually):

1. For each datapoint, calculate actual value minus predicted value.
2. Square each distance value.
3. Sum together all these squared values.
4. Divide the sum by the total number of data points.
5. Take the square root of the previous answer.

Squaring the difference is useful because it helps penalize the model for larger errors. Taking the square root at the end changes the error value back into the same units as your target.

R^2

Unlike RMSE, the higher the score here, the better. R^2 takes your model and compares it to a much more basic model – what would happen if you simply predicted the average value of your target variable for every single data point? In other words, R^2 evaluates how much better your model is compared to predicting that average value for everything.

Optional additional resource: article on regression metrics

Coding evaluation metrics

Alrighty, let's dive back in! We'll use sklearn's metric methods to make this simple.

First, some imports:

```
from sklearn.metrics import precision_score, recall_score, accuracy_score
```

Then...

```
precision_score(y_test, y_predicted)
```

Let's analyze the predictions:

```
pd.DataFrame(y_predicted).value_counts()
```

Let's look at a few other measures of how well we're doing:

```
recall_score(y_test, y_predicted)
```

And now accuracy:

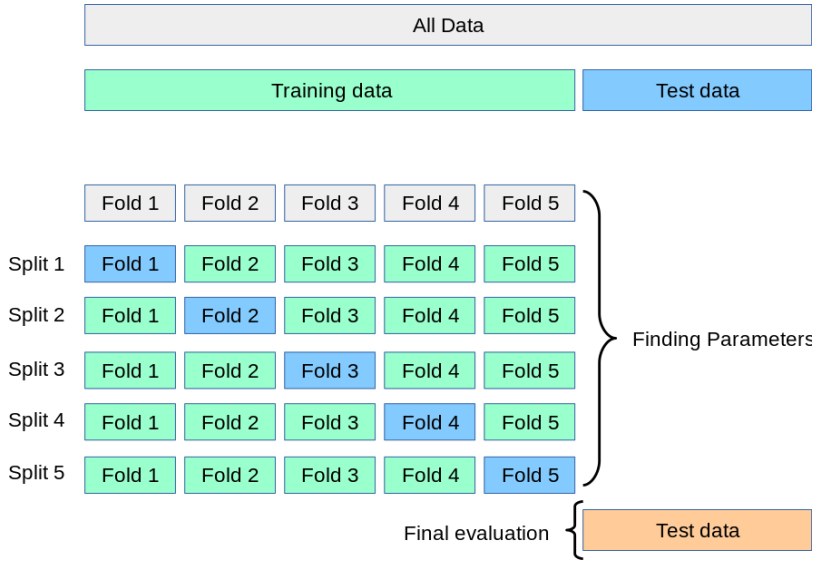
```
accuracy_score(y_test, y_predicted)
```

So accuracy is the highest, and we're relatively balanced on recall & precision. Let's optimize our model and see if we can make this better.

Optimizing your model

One thing we can do is gridsearch multiple models with different parameters (i.e. changes in setup, we won't go into those). But remember, we don't want to optimize using our test set, otherwise we won't know how the model will perform on unseen data.

So instead, we'll use **cross validation**, meaning we will split our train set even further, this time into mini train sets and mini test sets. We can make these splits multiple times and test out our different models on this mini splits to figure out which model and parameters may work best. But we will not touch that final test set at all!



(Source)

We need to make one quick change first – datto works on pipelines, not single models. Pipelines allow you to do multiple transformations to your data before running a model. We won't go into detail of those other steps in this course, but instead we'll set up our model in pipeline format quick to make it compatible with datto's methods.

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([("model", LogisticRegression()),])
```

Then, we'll run the model_testing function. This function will test several metrics, but we can choose one to optimize most. Let's do precision, because we've decided that it's more important to be correct and selective in labeling survivors.

```
tm = dt.TrainModel()
tm.model_testing(X_train_combined, y_train, pipeline, "classification", "precision")
```

Note that this will take a few minutes to run.

Here you'll get your results. I got the results that a MLP Classifier worked best (you may get slightly different results), so I'm going to rerun it on the test set using this new classifier plus the best parameters listed and get my final results.

```
from sklearn.neural_network import MLPClassifier

pipeline = Pipeline([("model", MLPClassifier(activation='identity', alpha=0.1))])

pipeline.fit(X_train_combined, y_train)
y_predicted = pipeline.predict(X_test_combined)

# You can use the same scoring methods as before,
# but this datto method gives more detailed results
mr = dt.ModelResults()
mr.score_final_model(
    "classification", X_train_combined, y_train, X_test_combined, y_test, pipeline
)
```

Looks like the model is doing just a bit better now!

Once you have a model that is working well, you can simple save it somewhere (such as S3) and then use it to predict incoming data somewhere, such as within a micro-service. You'll need to load the model upon starting the service, and then you can feed it new data to predict one by one. Make sure to fit your model on your data before saving it, and to feed it the exact same columns of features in order to make predictions. To do this, you'll need to rerun all the same steps that you did when cleaning your data previously.

Additional resources

Through this course, you've seen ways to implement datto methods that help with the machine learning model development process. If you'd like to see more examples of how to use this package and additional methods, check out this Juypter notebook.

Other helpful packages:

- Hypothesis - automatically tests several data structures in each test
- Altair - creates HTML interactive visualizations
- Featuretools - generates additional features for use in your model
- snorkel - assists in creating labels for unlabeled data
- fancyimpute - many methods for better filling null values
- Modin - faster computing in Pandas

Learning resources:

- SQL course: https://www.codecademy.com/learn/learn-sql
- Python course: https://www.codecademy.com/learn/learn-python
- Pandas tutorial: https://www.youtube.com/watch?v=w26x-z-BdWQ&t=1217s
- Linear algebra videos: https://www.youtube.com/watch?v=kjBOesZCoqc&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab
- Calculus concepts videos: https://www.youtube.com/watch?v=WUvTyaaNkzM&list=PLZHQObOWTQDMsr9K-rj53DwVRMYO3t5Y
- Probability distribution videos (all discrete & continuous videos): http://www.jbstatistics.com/discrete-probability-distributions/

- Machine learning course (free without the official certificate): https://www.coursera.org/learn/machine-learning
- Machine learning videos by topic: https://brohrer.github.io/blog.html
- Teaching materials I developed: https://github.com/kristiewirth/ds-teaching-materials
- Conversion rates articles: https://erikbern.com/2017/05/23/conversion-rates-you-are-most-likely-computing-them-wrong.html
- HTML course: https://www.codecademy.com/learn/learn-html
- Bash/shell course: https://www.codecademy.com/learn/learn-the-command-line
- Regular expressions: https://www.dataquest.io/blog/regular-expressions-data-scientists/
- Open-source data science masters (many many resources here): http://datasciencemasters.org/

Conclusion

I hope this guide has given you a better idea of what machine learning is, how it works, and how you can use it to create simple models!

To recap some important concepts:

- You can use machine learning to either predict something (supervised learning) or to create meaningful groups (unsupervised learning).
- Always start with exploring your data – see what values you have in different columns, check data types, check for null values, etc.
- To prepare your data for modeling, you'll need to fill null values, split into a train set and a test set, transform text data to word frequencies, and transform categorical data into numerical columns
- Your features are the information used to predict something, and your target is what you are trying to predict
- Classifiers are used when you have a categorical target, regressors are used when you have a numerical target
- Accuracy as a way to evaluate your model can be misleading – think about your use case and decide if precision or recall is a better fit
- Never learn anything from your test set – split your train set into smaller pieces and use gridsearching to test out different models and parameters to optimize your model

Best of luck working on your own models! Please share any feedback with us about how this course went.