Machine Learning: The DL on ML

This replicates, in a jupyter notebook, the code found in Chapter 7 (“Machine Learning: The DL[1] on ML”) of Meredith Broussard’s **Artificial Unintelligence: How Computers Misunderstand the World** (MIT Press, 2018, https://mitpress.mit.edu/books/artificial-unintelligence (https://mitpress.mit.edu/books/artificial-unintelligence)). I have also added an additional section on “Overfitting” at the end.

As Broussard notes, her treatment is a simplified version of the Kaggle Python Tutorial on Machine Learning at Datacamp (https://www.datacamp.com/community/open-courses/kaggle-python-tutorial-on-machine-learning (https://www.datacamp.com/community/open-courses/kaggle-python-tutorial-on-machine-learning)), but I think it is also helpful to have a companion following the book chapter. I also address one subtle but enormously consequential mistake Broussard makes: reporting training accuracy as test accuracy. This may be a typo, but even as a typo it is a significant oversight. Note that the datacamp tutorial discusses overfitting on its 6th lesson of the second unit, https://campus.datacamp.com/courses/kaggle-python-tutorial-on-machine-learning/predicting-with-decision-trees?ex=6 (https://campus.datacamp.com/courses/kaggle-python-tutorial-on-machine-learning/predicting-with-decision-trees?ex=6), although this is cursory.

The mistake notwithstanding, I think Broussard’s chapter is the single best overview of machine learning for a non-technical audience in terms of (hopefully) demystifying machine learning, and providing an alternative narrative to those of commonly made public claims. The commentary she provides while going through the example emphasizes

- the **tedium** of doing machine learning,
- the surprising simplicity (and even ”stupidity”) of much of it, and
- some fundamental limitations in data and modeling (such as through her point about numbering of rafts; this can be phrased as a problem of a latent variable being causal, but so can many cases of features that predict well in data from one case but will fail to generalize).

This notebook is to help anybody who might want to move from that treatment to a technical one, or perhaps see the visual artifact of what doing machine learning/data science would look like, in context (on an analyst's computer, rather than filtered through the book’s typography). To actually run this notebook, you will unfortunately need to go through a standard but still laborious process of installing multiple things on your computer.

There is one largely unimportant mistake:

- Feature importance is NOT the same as statistical significance! They frequently coincide, but it is possible to have an important feature that is not statistically significant, and conversely, it is possible to have a statistically significance variable low in feature importance. Feature importance has to do with the (normalized) size of an effect, whereas statistical significance has to do with the amount of variance of that effect across observations, given the sample size. This difference is further explored in the literature around “post-selection inference.”

But there is one major mistake:

- The test accuracy is .72, NOT .97 as Broussard claims!

> page 114: “We can write the predictions to a CSV file called `my_solution_one.csv`, upload the file to DataCamp, and verify that our predictions were 97 percent accurate. Ta-da! We just did machine learning.”

The training accuracy is 0.9776 (also, this should round up to 0.98, not down to 0.97, but that’s really not important here), but submitting the ‘predicted’ values to Kaggle gives an accuracy of 0.72248. **The fact that test performance is always worse than training performance is critical**, as it illustrates the common problem of **overfitting**: I discuss

this more at the bottom, but this is important because overfitting means that machine learning, in the hypothetical settings of data analysis (like in this notebook), can easily seem to perform far better than it actually will when deployed in the world.

Additionally, there is one typographic error: all of the python code has commas inside quotes, perhaps the result of a careless “find+replace all” of <",> with <,">. Be aware of this if you copy code from the book, but it’s fixed in the copy below.

In the book chapter, there are some redundant commands (in the text, each instance has a different explanation); I kept these for faithfulness to the chapter. These commands follow `train.describe()` (inputs `[5]` to `[10]`).

[1] “DL” here stands for “down-low” and here means something like “secrets”. Now mainstream, I believe it comes from appropriating queer black slang.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn import tree, preprocessing
```

In [2]:
```
train_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/train.csv"
train = pd.read_csv(train_url)
test_url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/test.csv"
test = pd.read_csv(test_url)
```

In [3]:
```
print(train.head())
```

```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
4      0            373450   8.0500   NaN        S
```

In [4]:
```
print(test.head())
```

```
   PassengerId  Pclass                                          Name     Sex  \
0          892       3                              Kelly, Mr. James    male
1          893       3              Wilkes, Mrs. James (Ellen Needs)  female
2          894       2                     Myles, Mr. Thomas Francis    male
3          895       3                              Wirz, Mr. Albert    male
4          896       3  Hirvonen, Mrs. Alexander (Helga E Lindqvist)  female

    Age  SibSp  Parch   Ticket     Fare Cabin Embarked
0  34.5      0      0   330911   7.8292   NaN        Q
1  47.0      1      0   363272   7.0000   NaN        S
2  62.0      0      0   240276   9.6875   NaN        Q
3  27.0      0      0   315154   8.6625   NaN        S
4  22.0      1      1  3101298  12.2875   NaN        S
```

In [5]:
```
train.describe()
```

Out[5]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [6]:
```
train["Pclass"].value_counts()
```

Out[6]:
```
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

In [7]:
```
train["Survived"].value_counts() # Note: redundant with two lines down
```

Out[7]:
```
0    549
1    342
Name: Survived, dtype: int64
```

In [8]:
```
print(train["Survived"].value_counts(normalize = True)) # Note: redundant with tw
o lines down
```

```
0    0.616162
1    0.383838
Name: Survived, dtype: float64
```

In [9]:
```
# Passengers that survived vs passengers that passed away
print(train["Survived"].value_counts())
```

```
0    549
1    342
Name: Survived, dtype: int64
```

In [10]:
```
# As proportions
print(train["Survived"].value_counts(normalize = True))
```

```
0    0.616162
1    0.383838
Name: Survived, dtype: float64
```

In [11]:
```
# Males that survived vs males that passed away
print(train["Survived"][train["Sex"] == 'male'].value_counts())
```

```
0    468
1    109
Name: Survived, dtype: int64
```

In [12]:
```
# Females that survived vs females that passed away
print(train["Survived"][train["Sex"] == 'female'].value_counts())
```

```
1    233
0     81
Name: Survived, dtype: int64
```

In [13]:
```
# Normalized male survival
print(train["Survived"][train["Sex"] == 'male'].value_counts(normalize=True))
```

```
0    0.811092
1    0.188908
Name: Survived, dtype: float64
```

In [14]:
```
# Normalized female survival
print(train["Survived"][train["Sex"] == 'female'].value_counts(normalize=True))
```

```
1    0.742038
0    0.257962
Name: Survived, dtype: float64
```

In [15]:
```
train["Age"] = train["Age"].fillna(train["Age"].median())
```

In [16]:
```
# Print the train data to see the available features
print(train)
```

	PassengerId	Survived	Pclass	\
0	1	0	3	
1	2	1	1	
2	3	1	3	
3	4	1	1	
4	5	0	3	
5	6	0	3	
6	7	0	1	
7	8	0	3	
8	9	1	3	
9	10	1	2	
10	11	1	3	
11	12	1	1	
12	13	0	3	
13	14	0	3	
14	15	0	3	
15	16	1	2	
16	17	0	3	
17	18	1	2	
18	19	0	3	
19	20	1	3	
20	21	0	2	
21	22	1	2	
22	23	1	3	
23	24	1	1	
24	25	0	3	
25	26	1	3	
26	27	0	3	
27	28	0	1	
28	29	1	3	
29	30	0	3	
..	...	...	...	
861	862	0	2	
862	863	1	1	
863	864	0	3	
864	865	0	2	
865	866	1	2	
866	867	1	2	
867	868	0	1	
868	869	0	3	
869	870	1	3	
870	871	0	3	
871	872	1	1	
872	873	0	1	
873	874	0	3	
874	875	1	2	
875	876	1	3	
876	877	0	3	
877	878	0	3	
878	879	0	3	
879	880	1	1	
880	881	1	2	
881	882	0	3	
882	883	0	3	
883	884	0	2	
884	885	0	3	
885	886	0	3	
886	887	0	2	
887	888	1	1	
888	889	0	3	
889	890	1	1	

```
890          891         0       3

                                                    Name     Sex   Age  SibSp  \
0                                Braund, Mr. Owen Harris    male  22.0      1
1    Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                                Heikkinen, Miss. Laina  female  26.0      0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                              Allen, Mr. William Henry    male  35.0      0
5                                      Moran, Mr. James    male  28.0      0
6                               McCarthy, Mr. Timothy J    male  54.0      0
7                        Palsson, Master. Gosta Leonard    male   2.0      3
8    Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)  female  27.0      0
9                  Nasser, Mrs. Nicholas (Adele Achem)  female  14.0      1
10                     Sandstrom, Miss. Marguerite Rut  female   4.0      1
11                            Bonnell, Miss. Elizabeth  female  58.0      0
12                      Saundercock, Mr. William Henry    male  20.0      0
13                         Andersson, Mr. Anders Johan    male  39.0      1
14                Vestrom, Miss. Hulda Amanda Adolfina  female  14.0      0
15                    Hewlett, Mrs. (Mary D Kingcome)   female  55.0      0
16                                Rice, Master. Eugene    male   2.0      4
17                        Williams, Mr. Charles Eugene    male  28.0      0
18   Vander Planke, Mrs. Julius (Emelia Maria Vande...  female  31.0      1
19                              Masselmani, Mrs. Fatima  female  28.0      0
20                                 Fynney, Mr. Joseph J    male  35.0      0
21                               Beesley, Mr. Lawrence    male  34.0      0
22                         McGowan, Miss. Anna "Annie"  female  15.0      0
23                        Sloper, Mr. William Thompson    male  28.0      0
24                       Palsson, Miss. Torborg Danira  female   8.0      3
25   Asplund, Mrs. Carl Oscar (Selma Augusta Emilia...  female  38.0      1
26                              Emir, Mr. Farred Chehab    male  28.0      0
27                       Fortune, Mr. Charles Alexander    male  19.0      3
28                       O'Dwyer, Miss. Ellen "Nellie"  female  28.0      0
29                                  Todoroff, Mr. Lalio    male  28.0      0
..                                                  ...     ...   ...    ...
861                          Giles, Mr. Frederick Edward    male  21.0      1
862  Swift, Mrs. Frederick Joel (Margaret Welles Ba...  female  48.0      0
863                  Sage, Miss. Dorothy Edith "Dolly"  female  28.0      8
864                               Gill, Mr. John William    male  24.0      0
865                              Bystrom, Mrs. (Karolina)  female  42.0      0
866                         Duran y More, Miss. Asuncion  female  27.0      1
867                Roebling, Mr. Washington Augustus II    male  31.0      0
868                          van Melkebeke, Mr. Philemon    male  28.0      0
869                     Johnson, Master. Harold Theodor    male   4.0      1
870                                  Balkic, Mr. Cerin    male  26.0      0
871   Beckwith, Mrs. Richard Leonard (Sallie Monypeny)  female  47.0      1
872                              Carlsson, Mr. Frans Olof    male  33.0      0
873                          Vander Cruyssen, Mr. Victor    male  47.0      0
874                Abelson, Mrs. Samuel (Hannah Wizosky)  female  28.0      1
875                     Najib, Miss. Adele Kiamie "Jane"  female  15.0      0
876                        Gustafsson, Mr. Alfred Ossian    male  20.0      0
877                                 Petroff, Mr. Nedelio    male  19.0      0
878                                   Laleff, Mr. Kristo    male  28.0      0
879      Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)  female  56.0      0
880       Shelley, Mrs. William (Imanita Parrish Hall)  female  25.0      0
881                                  Markun, Mr. Johann    male  33.0      0
882                         Dahlberg, Miss. Gerda Ulrika  female  22.0      0
883                       Banfield, Mr. Frederick James    male  28.0      0
884                              Sutehall, Mr. Henry Jr    male  25.0      0
885                Rice, Mrs. William (Margaret Norton)  female  39.0      0
886                                Montvila, Rev. Juozas    male  27.0      0
887                         Graham, Miss. Margaret Edith  female  19.0      0
888           Johnston, Miss. Catherine Helen "Carrie"  female  28.0      1
```

889				Behr, Mr. Karl Howell	male	26.0	0		
890				Dooley, Mr. Patrick	male	32.0	0		

	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
1	0	PC 17599	71.2833	C85	C
2	0	STON/O2. 3101282	7.9250	NaN	S
3	0	113803	53.1000	C123	S
4	0	373450	8.0500	NaN	S
5	0	330877	8.4583	NaN	Q
6	0	17463	51.8625	E46	S
7	1	349909	21.0750	NaN	S
8	2	347742	11.1333	NaN	S
9	0	237736	30.0708	NaN	C
10	1	PP 9549	16.7000	G6	S
11	0	113783	26.5500	C103	S
12	0	A/5. 2151	8.0500	NaN	S
13	5	347082	31.2750	NaN	S
14	0	350406	7.8542	NaN	S
15	0	248706	16.0000	NaN	S
16	1	382652	29.1250	NaN	Q
17	0	244373	13.0000	NaN	S
18	0	345763	18.0000	NaN	S
19	0	2649	7.2250	NaN	C
20	0	239865	26.0000	NaN	S
21	0	248698	13.0000	D56	S
22	0	330923	8.0292	NaN	Q
23	0	113788	35.5000	A6	S
24	1	349909	21.0750	NaN	S
25	5	347077	31.3875	NaN	S
26	0	2631	7.2250	NaN	C
27	2	19950	263.0000	C23 C25 C27	S
28	0	330959	7.8792	NaN	Q
29	0	349216	7.8958	NaN	S
..	...	...	...	...	...
861	0	28134	11.5000	NaN	S
862	0	17466	25.9292	D17	S
863	2	CA. 2343	69.5500	NaN	S
864	0	233866	13.0000	NaN	S
865	0	236852	13.0000	NaN	S
866	0	SC/PARIS 2149	13.8583	NaN	C
867	0	PC 17590	50.4958	A24	S
868	0	345777	9.5000	NaN	S
869	1	347742	11.1333	NaN	S
870	0	349248	7.8958	NaN	S
871	1	11751	52.5542	D35	S
872	0	695	5.0000	B51 B53 B55	S
873	0	345765	9.0000	NaN	S
874	0	P/PP 3381	24.0000	NaN	C
875	0	2667	7.2250	NaN	C
876	0	7534	9.8458	NaN	S
877	0	349212	7.8958	NaN	S
878	0	349217	7.8958	NaN	S
879	1	11767	83.1583	C50	C
880	1	230433	26.0000	NaN	S
881	0	349257	7.8958	NaN	S
882	0	7552	10.5167	NaN	S
883	0	C.A./SOTON 34068	10.5000	NaN	S
884	0	SOTON/OQ 392076	7.0500	NaN	S
885	5	382652	29.1250	NaN	Q
886	0	211536	13.0000	NaN	S
887	0	112053	30.0000	B42	S

```
888      2        W./C. 6607   23.4500          NaN        S
889      0            111369   30.0000         C148        C
890      0            370376    7.7500          NaN        Q

[891 rows x 12 columns]
```

In [17]:
```
# Create the target and features numpy arrays: target, features_one
target = train["Survived"].values
```

In [18]:
```
# Preprocess
encoded_sex = preprocessing.LabelEncoder()
```

In [19]:
```
# Convert into numbers
train.Sex = encoded_sex.fit_transform(train.Sex)
features_one = train[["Pclass", "Sex", "Age", "Fare"]].values
```

In [20]:
```
# Fit the first decision tree: my_tree_one
my_tree_one = tree.DecisionTreeClassifier()
my_tree_one = my_tree_one.fit(features_one, target)
```

In [21]:
```
# Look at the importance and score of the included features
print(my_tree_one.feature_importances_)
```

```
[ 0.12315342  0.31274009  0.23735946  0.32674702]
```

In [22]:
```
print(my_tree_one.score(features_one, target))
```

```
0.977553310887
```

In [23]:
```
# Fill any missing fare values with the median fare
test["Fare"] = test["Fare"].fillna(test["Fare"].median())
```

In [24]:
```
# Fill any missing age values with the median age
test["Age"] = test["Age"].fillna(test["Age"].median())
```

In [25]:
```
# Preprocess
test_encoded_sex = preprocessing.LabelEncoder()
test.Sex = test_encoded_sex.fit_transform(test.Sex)
```

In [26]:
```
# Extract important features from the test set: Pclass, Sex, Age, and Fare
test_features = test[["Pclass", "Sex", "Age", "Fare"]].values
print('These are the features:\n')
print(test_features)
```

```
These are the features:

[[  3.       1.      34.5      7.8292]
 [  3.       0.      47.       7.    ]
 [  2.       1.      62.       9.6875]
 ...,
 [  3.       1.      38.5      7.25  ]
 [  3.       1.      27.       8.05  ]
 [  3.       1.      27.      22.3583]]
```

In [27]:
```
# Make a prediction using the test set and print
my_prediction = my_tree_one.predict(test_features)
print('This is the prediction:\n')
print(my_prediction)
```

```
This is the prediction:

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

In [28]:
```
# Create a data frame with two columns: PassengerId & Survived
# Survived contains the model's prediction
PassengerId =np.array(test["PassengerId"]).astype(int)
my_solution = pd.DataFrame(my_prediction, PassengerId, columns = ["Survived"])
print('This is the solution in toto:\n')
print(my_solution)
```

This is the solution in toto:

	Survived
892	0
893	0
894	1
895	1
896	1
897	0
898	0
899	0
900	1
901	0
902	0
903	0
904	1
905	1
906	1
907	1
908	0
909	1
910	1
911	0
912	0
913	1
914	1
915	0
916	1
917	0
918	1
919	1
920	1
921	0
...	...
1280	0
1281	0
1282	0
1283	1
1284	0
1285	0
1286	0
1287	1
1288	0
1289	1
1290	0
1291	0
1292	1
1293	0
1294	1
1295	1
1296	0
1297	0
1298	0
1299	0
1300	1
1301	1
1302	1
1303	1
1304	0
1305	0
1306	1

```
1307          0
1308          0
1309          0

[418 rows x 1 columns]
```

In [29]:
```
# Check that the data frame has 418 entries
print('This is the solution shape:\n')
print(my_solution.shape)
```

```
This is the solution shape:

(418, 1)
```

In [30]:
```
# Write the solution to a CSV file with the name my_solution.csv
my_solution.to_csv("my_solution_one.csv", index_label = ["PassengerId"])
```

Now, you can submit this csv at https://www.kaggle.com/c/titanic/leaderboard (https://www.kaggle.com/c/titanic/leaderboard). This requires registering with Kaggle. If you do submit, you should get an accuracy of 0.72248, like I did:

9112	**Momin Malik**		0.72248

Overfitting

Here I go further into the critical point that the training accuracy is different from the test accuracy, and specifically, the test accuracy is almost always going to be lower than the training accuracy (if that ever seems to not be the case, it's more likely something is wrong with the code or data).

What is overfitting? If we imagine that there is some underlying 'signal' in our data, but also 'noise', then overfitting is when our models pick up on the noise rather than the signal (yes, machine learning and statistics involve an implicit metaphysical commitment—with consequences for how we interact with the world—although that is a much larger discussion). The noise, of course, never stays constant, and so when we see new data, with different noise, our previous model will fail. In the metaphorical language of machine learning, the model 'memorizes' the data and then can't generalize.

The technique of cross-validation attempts to address this by splitting the data into two parts, 'training' and 'testing' (the ratio of splitting the data into training and test may be 1:1, or maybe 4:1, or 9:1). The idea is that the signal will be the same across the split, but not the noise; by fitting our model on the training set and then comparing it against the test set, we can get a more honest assessment of whether or not we have captured some signal, or if we are only picking up noise.

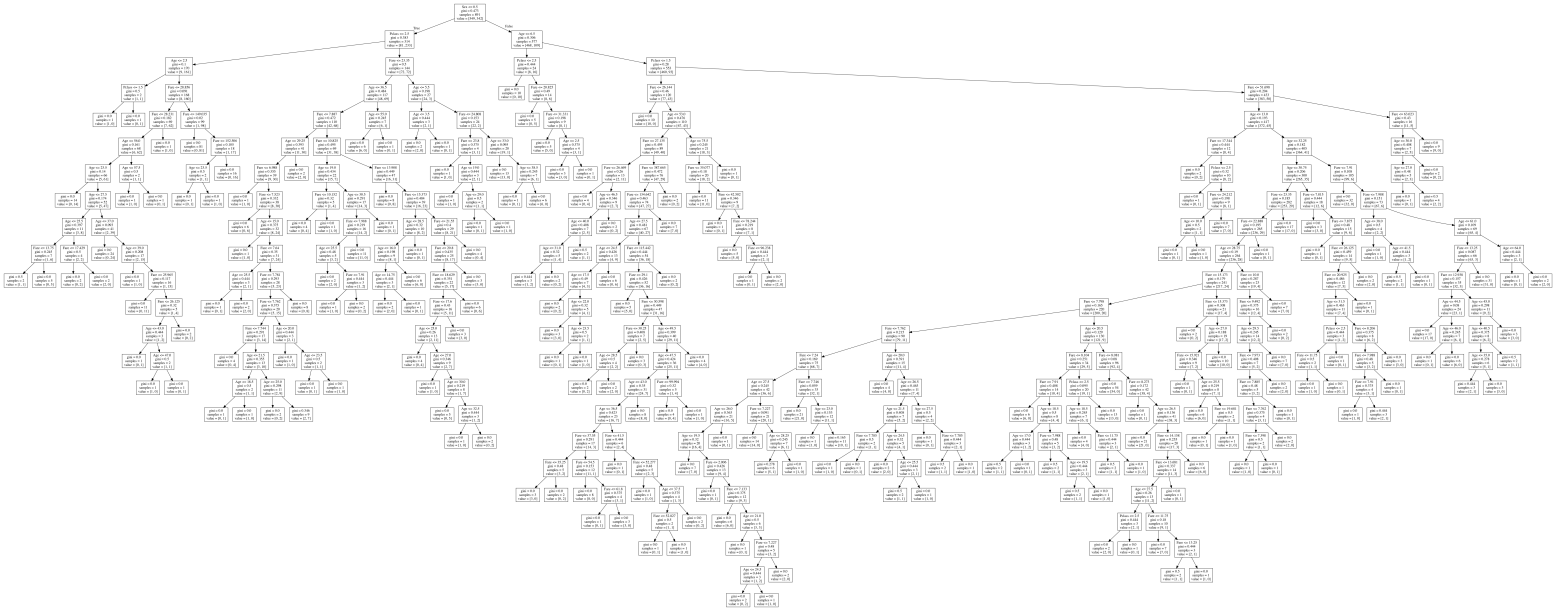
Note that even test accuracy is not reliable as an estimate of how well a model will do when deployed: there are ways that the performance on truly new data might again be worse than even the test performance (if there was sampling bias towards observations with a strong signal; if there are temporal, spatial, network, or other dependencies in the data that the data splitting does not or cannot control for; and if we set aside data to serve only as a test set but try so many models that we end up overfitting via a distribution over models).

Overfitting has much to do with how complicated a model is; complicated models are far more likely to be picking up noise than reflecting complexity in the world. For this reason, simple models often work better than complicated ones.

We can see how a complicated model can lead to failure by visualizing the decision tree that gave the .98 accuracy.

In order to plot the tree, you will need to install graphviz, e.g., `conda install python-graphviz` in the command line. Warning: this can be finicky. I exported the tree as a file in order to convert to an image with `tree.export_graphviz(my_tree_one, out_file='tree.dot', feature_names=["Pclass", "Sex", "Age", "Fare"])`, and that image is what is shown here.

In [31]:
```
from graphviz import Source
Source(tree.export_graphviz(my_tree_one, out_file='None', feature_names=["Pclass"
, "Sex", "Age", "Fare"]))
```



We see that the tree has dozens of splits, maybe even more than a hundred, for only four variables!

To zoom in, you can right click and open the above image in a new tab, or look at the pdf at https://github.com/momin-malik/guides/blob/master/tree.pdf (https://github.com/momin-malik/guides/blob/master/tree.pdf).

We can also see that the terminal nodes (also called 'leaf nodes') at the end of branching often contain only 1 or 2 samples: this is what is called "memorizing" the data.

A better approach is to use some limits on how complex the tree can be. Specifically, I use the default parameters (https://www.rdocumentation.org/packages/tree/versions/1.0-39/topics/tree.control) of the `R` implementation of decision trees in the `tree` package. These parameters are:

- `mincut = 5`
- `minsize = 10`
- `mindev = 0.01`

In the `scikitlearn` implementation of decision trees (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html), the corresponding parameters are

- `min_samples_split` (default = 2)
- `min_samples_leaf` (default = 1)
- `min_impurity_decrease` (default = 0.)

Note that the datacamp tutorial at https://campus.datacamp.com/courses/kaggle-python-tutorial-on-machine-learning/predicting-with-decision-trees?ex=6 (https://campus.datacamp.com/courses/kaggle-python-tutorial-on-machine-learning/predicting-with-decision-trees?ex=6) uses the parameters `max_depth = 10, min_samples_split = 5, random_state = 1`, although there is no explanation of where these come from.

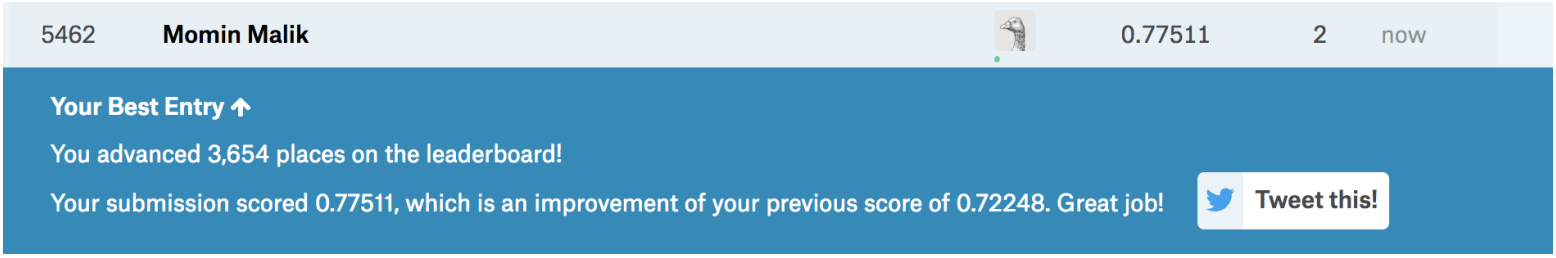
Granted, there is no explanation in the `R` documentation either of why the default parameters are what they are, although the fact that they are the defaults here reflects a philosophy that the defaults should encode what works, rather than the base form of the model. As for where these come from—these settings are known as "tuning parameters", also known as "hyperparameters" (and, confusingly, we sometimes talk of "tuning the hyperparameters") in that they 'tune' how the model is fit to data but (unlike **model** parameters, like the feature importance from decision trees) are not something we interpret to try and understand something about the data. Default values of tuning parameters sometimes have theoretical reasons (like histogram bin width), but in this case they probably are the result of the trial and error of dozens of the heavy users who wrote the `R` code in question, a sort of folk wisdom (which may be reliable, just without theoretical justification). In practice, we could optimize these parameters as well, by splitting the data into not only training and test, but training, validation, and test; we would fit multiple decision trees, with different combinations of tuning parameters, on the training set, see which leads to the highest accuracy on the validation set, choose that one as our model, and then see how well it works by testing on the test set. The additional split into a validationset is to prevent overfitting on the tuning parameters, which is also possible and easy to fall into. But often, the choice of tuning parameters is very much of a subjective process, or even a "dark art," something without justification but that can determine whether or not a model is useless or miraculous (although if tuning parameters are not chosen systematically, it can lead to overfitting).

In [32]:
```
# Fit the second decision tree: my_tree_two
my_tree_two = tree.DecisionTreeClassifier(min_samples_split = 5,
                                          min_samples_leaf = 10,
                                          min_impurity_decrease = 0.01)
my_tree_two = my_tree_two.fit(features_one, target)
```

```
In [33]: print(my_tree_two.score(features_one, target))
         0.819304152637
```

The training accuracy is substantially lower! It seems like we did worse, right? Well, using the same `test_features`, we can make another set of values to submit to Kaggle.

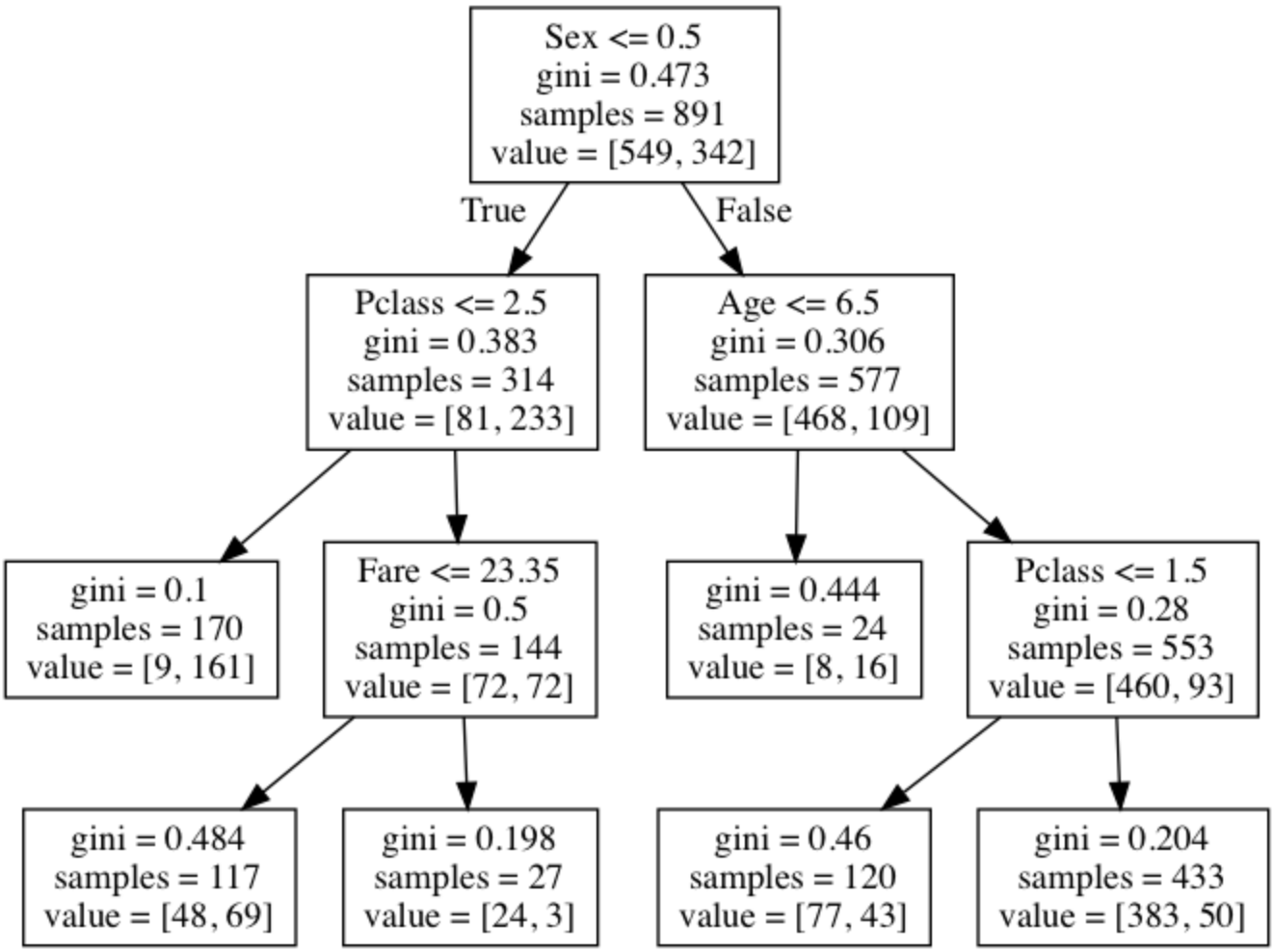
```
In [34]: my_prediction_two = my_tree_two.predict(test_features)
         PassengerId = np.array(test["PassengerId"]).astype(int)
         my_solution_two = pd.DataFrame(my_prediction_two, PassengerId, columns = ["Surviv
         ed"])
         my_solution_two.to_csv("my_solution_two.csv", index_label = ["PassengerId"])
```



Our test accuracy was actually **higher**, at 0.7751! (While we advanced 3,654 places in the leaderboard, most entries are from people following the Datacamp tutorial, various units of which deterministically give the same test accuracy; so, the number of people we jumped will mostly be counting how many people completed certain units).

There are ways to statistically test if the improvement from 0.72248 to 0.7751 is “significant” or not, but that’s a longer process. Rest assured, this increase is indeed significant.

What does the “better” tree looks like? Again, I export with `tree.export_graphviz(my_tree_two, out_file='tree2.dot', feature_names=["Pclass", "Sex", "Age", "Fare"])` and convert to an image. This time there’s no need for a pdf, as the tree is much simpler!



The far less complicated tree performs better on the test data than the complicated one. This illustrates overfitting, and how cross-validation, imperfect as it may be, is the first line of defense against machine learning going awry.

End note 1: `R` versus `python`

I do prefer `R` to `python` for data analysis; I recognize there are many areas where `python` is far better (e.g., production settings, integration with other systems, running on servers, deep learning), but I think it is meaningful that the default parameters in the implementation of decision trees in `R` take much better care of the user than do the default parameters of decision trees in `scikitlearn`. At a minimum, my knowing how to run the same model in both `R` and `python` frequently proves useful for checking over my work, that of collaborators, and of others I find.

End note 2: The social context of decision trees

Decision trees are fascinating historically and in the philosophy of statistics; if you are interested in learning more, there is fortunately a fantastic article about them by historian of science Matt Jones, "How We Became Instrumentalists (Again): Data Positivism since World War II," **Historical Studies in the Natural Sciences** 48, no. 5 (November 1, 2018): 673–84, available at http://www.columbia.edu/~mj340/HSNS4805_12_Jones.pdf (http://www.columbia.edu/~mj340/HSNS4805_12_Jones.pdf).