

# Your First ML Model

## *Teaching a model to read DNA*

The full notebook // @hinnaazeejah

---

This is the companion to the reel. Every cell below is the exact code from the video, with an explanation of what each line is doing and why it matters.

Five cells. No installs. No GPU. It runs in Google Colab on cafe wifi in under two minutes.

**Start here:** Open the finished notebook in Colab. [Open the notebook](#)

### **Make it yours before you run anything**

The link opens in view only mode, so the first thing to do is make your own copy. In Colab, go to File, then Save a copy in Drive. That copy is yours to edit, break, and rerun as many times as you want.

Then run the cells in order from top to bottom. Let each one finish before starting the next.

---

### **What you are building**

A model that looks at a stretch of E. coli DNA and predicts whether it is a promoter, which is the spot on the strand where the cell starts reading a gene. Think of it as the start switch.

You do not need to know any biology for this. To the model, DNA is just a string of four letters: a, c, g, t. Everything you learn here transfers directly to text, emails, reviews, and any other language data.

### **Why not the usual beginner project**

Almost every first ML project is the same handful of datasets, and recruiters have seen them a thousand times. This one is the same difficulty and the same amount of code, but it is real biological data, and it teaches you the single most important idea in machine learning: models only ever see numbers.

---

## Cell 1 // Load the DNA

```
import pandas as pd

url = ("https://archive.ics.uci.edu/ml/machine-learning-databases/"
      "molecular-biology/promoter-gene-sequences/promoters.data")

df = pd.read_csv(url, header=None, names=['label', 'name', 'sequence'])

sequences = df['sequence'].str.replace(r'\s', '', regex=True).str.lower()
labels = (df['label'].str.strip() == '+').astype(int)

print(len(sequences), "sequences")
print(sequences[0])
```

### What you should see

106 sequences, followed by one long string of letters.

### What is happening

- The dataset loads straight from a URL, so there is nothing to download or upload.
- Each row has a label, a name, and a 57 character DNA sequence.
- The replace step strips out tabs and spaces so you are left with clean letters only.
- The labels line turns a plus sign into 1 and a minus sign into 0, because models need numbers, not symbols.

**If the URL times out:** Open it in your browser, save the file, drag it into the Colab file panel on the left, and change the url line to "promoters.data".

---

## Cell 2 // Turn letters into numbers

This is the most important cell in the notebook.

```
from sklearn.feature_extraction.text import CountVectorizer

def to_kmers(seq, k=4):
    return ' '.join(seq[i:i+k] for i in range(len(seq) - k + 1))

kmer_text = sequences.apply(to_kmers)

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(kmer_text)
```

```
print("before:", sequences[0][:40])
print("after: ", X.toarray()[0][:40])
print("shape:", X.shape)
```

### What you should see

A line of letters, a line of numbers, and shape: (106, 256).

### What is happening

A model cannot read letters. It can only do math on numbers. So you have to convert the sequence into a numeric form, and the way you do that conversion is the whole skill.

- A k-mer is just a chunk of k letters. With k=4, the sequence tactag becomes tact, acta, ctag, and so on.
- The to\_kmers function slides a window across the sequence and returns those chunks separated by spaces, so the DNA now looks like a sentence made of four letter words.
- CountVectorizer then counts how many times each chunk appears. Every sequence becomes a row of 256 counts, one for every possible four letter combination.
- That is why the shape is 106 by 256. One row per sequence, one column per possible chunk.

**The transferable part:** This is the exact same technique used on written text. Swap DNA for movie reviews and k-mers for words, and you have just built the front half of a sentiment classifier. The idea generalizes; memorize it.

### Try this after

Change k=4 to k=3 or k=6 and rerun. Watch the shape change and watch the accuracy change with it. Larger k means more columns and more specificity, but with only 106 sequences it starts to hurt.

---

## Cell 3 // Split the data

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
```

```
X, labels, test_size=0.2, random_state=42, stratify=labels
)

print(X_train.shape[0], "to study")
print(X_test.shape[0], "to be tested on...")
```

### What you should see

84 to study and 22 to be tested on....

### Why this cell matters more than the model

You are holding back 20 percent of the data so the model never sees it during training. Then you grade the model on that held out portion.

If you skip this and test on the same data the model learned from, you will get a beautiful score that means absolutely nothing. It is the difference between passing an exam and being handed the answer key. This is the single most common beginner mistake and it is also the first thing an interviewer will check for.

- `test_size=0.2` holds back 20 percent.
- `random_state=42` makes the split reproducible, so you get the same result every run.
- `stratify=labels` keeps the same ratio of promoters to non promoters in both halves.

---

## Cell 4 // Build and train

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
```

### What is happening

That is the entire model. One line to create it, one line to train it, and it finishes in under a second.

Logistic regression learns a weight for every one of those 256 chunk counts. Chunks that show up more often in promoters get positive weights, chunks that show up more often in non promoters get negative ones. To make a prediction it adds up the weights and checks which side of zero it lands on.

max\_iter=1000 just gives the solver enough rounds to settle. Without it you may see a convergence warning.

---

## Cell 5 // Score it and make a prediction

```
accuracy = model.score(X_test, y_test)
print(f"accuracy: {accuracy:.1%}")

i = 0
guess = model.predict(X_test[i])[0]
truth = y_test.iloc[i]

print("model says:", "promoter" if guess else "not a promoter")
print("actually: ", "promoter" if truth else "not a promoter")
```

### What you should see

accuracy: 86.4%, which is 19 of the 22 test sequences correct, and a prediction printed in plain English.

### How to read that number honestly

86.4 percent on a test set of 22 sequences is a good result for five cells of code, and it is also a small number of examples. Three misses moves the score by roughly 14 points, so do not treat this as a precise measurement.

That caveat is not a weakness in your project. Being able to say out loud why your number is uncertain is what separates someone who ran a tutorial from someone who understands what they ran.

**If you see 100 percent:** Something is wrong. Check that you are scoring on X\_test and not X\_train. A perfect score on real data almost always means the model has seen the answers.

---

## Make it yours

The version above is the reel. These are the upgrades that turn it into something worth putting on a resume. Pick one, not all four.

- Run it at  $k=3, 4, 5$ , and  $6$  and plot accuracy against  $k$ . Now you have a chart and a finding, not just a script.
  - Use cross validation instead of one split, so your score is an average across several splits rather than one lucky draw.
  - Pull out the highest weighted chunks and look them up. Some of them correspond to real, documented promoter motifs, and noticing that is a genuinely impressive thing to talk about.
  - Swap logistic regression for a random forest and compare. Same data, same split, different model, one honest comparison.
- 

## Resume bullet template

*Fill in your own real numbers. Run it, read the output, use what you got.*

**Bullet:** Built a DNA sequence classifier in Python that predicts promoter regions from raw nucleotide sequences using  $k$ -mer feature extraction and logistic regression; achieved  $[x]\%$  accuracy on a held out test set and analyzed the effect of  $k$ -mer length on model performance.

Two things make that bullet land: the held out test set, which shows you know how to evaluate, and the  $k$ -mer analysis, which shows you did something beyond running the default.

---

## Common errors

### ConvergenceWarning

Harmless here. Raise `max_iter` if you want it to stop appearing.

### KeyError or a column mismatch in cell 1

The file has three comma separated fields. If the names list does not have exactly three entries, this breaks.

### The URL will not load

Download the file in your browser, drag it into the Colab file panel, and point the `read_csv` at the filename instead.

### Your accuracy is different from mine

Check that `random_state=42` is still in your split. Without it the split changes every run and so does the score.

---

### One last thing

Do not stop at running this. Change something, break it, and figure out why it broke. Change `k`. Remove the `stratify` argument. Test on the training data on purpose and watch the score jump to something suspicious.

The understanding comes from the breaking, not from the running.

**The notebook:** File, then Save a copy in Drive, then run it. [Open in Colab](#)

*go build something <3*

made by @hinnaazeejah · hinna