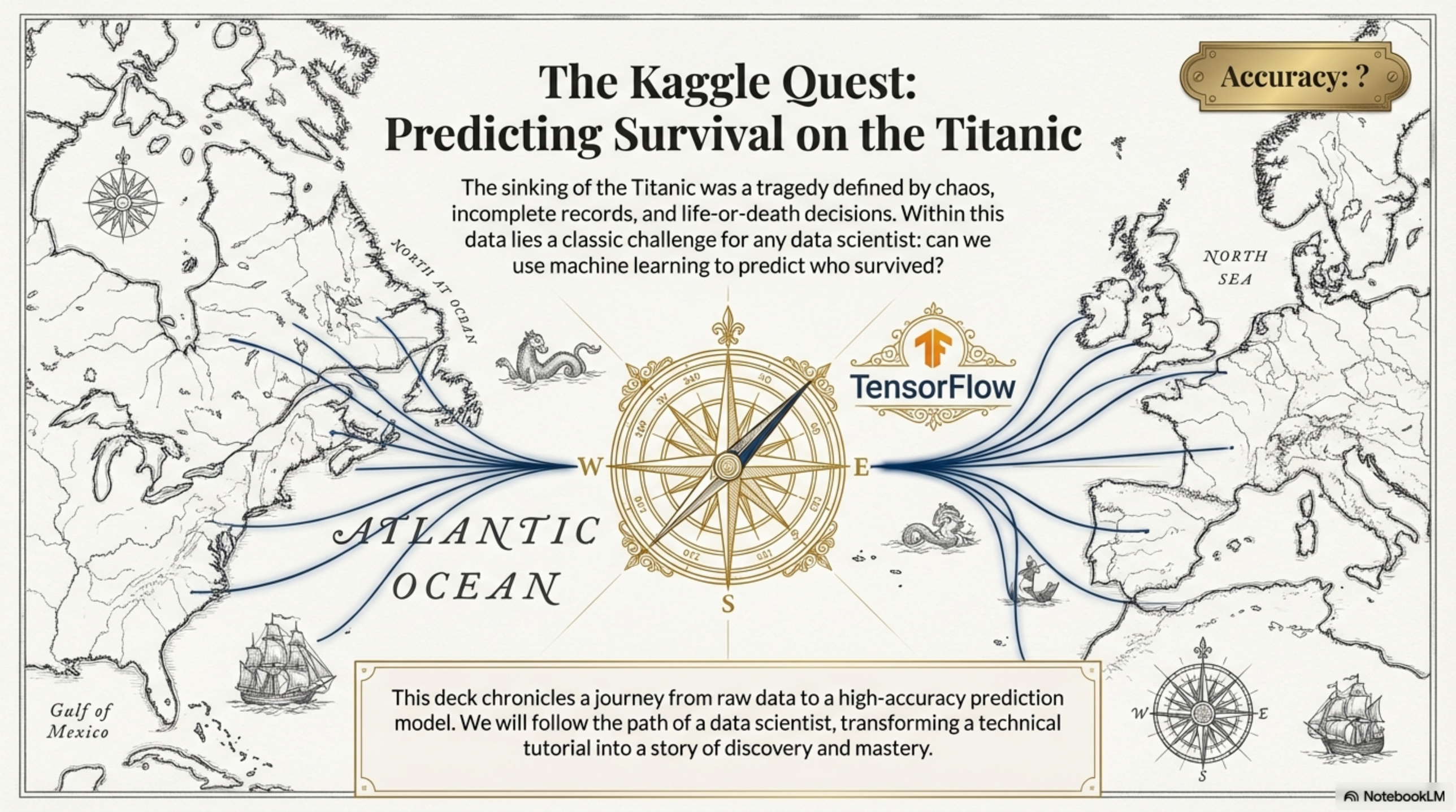
The Kaggle Quest: Predicting Survival on the Titanic

Accuracy: ?

The sinking of the Titanic was a tragedy defined by chaos, incomplete records, and life-or-death decisions. Within this data lies a classic challenge for any data scientist: can we use machine learning to predict who survived?



This deck chronicles a journey from raw data to a high-accuracy prediction model. We will follow the path of a data scientist, transforming a technical tutorial into a story of discovery and mastery.

Assembling Your Tools for the Voyage

kaggle

```
[1] import pandas as pd
    import tensorflow_decision_forests as tfdf
```

```
[2] train_df = pd.read_csv("train.csv")
    serving_df = pd.read_csv("test.csv")
```

```
[3] model = tfdf.keras.RandomForestModel()
```

The Dataset is Your Textbook

We begin with two tables of data: a `train_df` with known outcomes (the 'textbook' the model will study) and a `serving_df` with unknown outcomes (the 'final exam').

Features vs. The Label

Features: The clues about each passenger (Age, Sex, Pclass). These are the *questions*.
Label: The answer we want to predict (`Survived`). This is the *answer key*.

Why TensorFlow Decision Forests (TF-DF)? This powerful library simplifies many difficult parts of machine learning, like handling missing data and text, allowing us to focus on the story within the data.

The First Glimpse of the Data

PassengerId	Survived	Pclass	Name	Sex	Age
1	0	3	Braund, Mr. Owen Harris	male	22
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38
3	1	3	Heikkinen, Miss. Laina	female	26
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35
5	0	3	Allen, Mr. William Henry	male	35

This is our 'Label'—the ground truth we need to predict.

This is what the computer sees: a grid of passenger data. Our entire mission is to build a model that can look at the features on the left to predict the value in the `Survived` column.

Decoding the Label

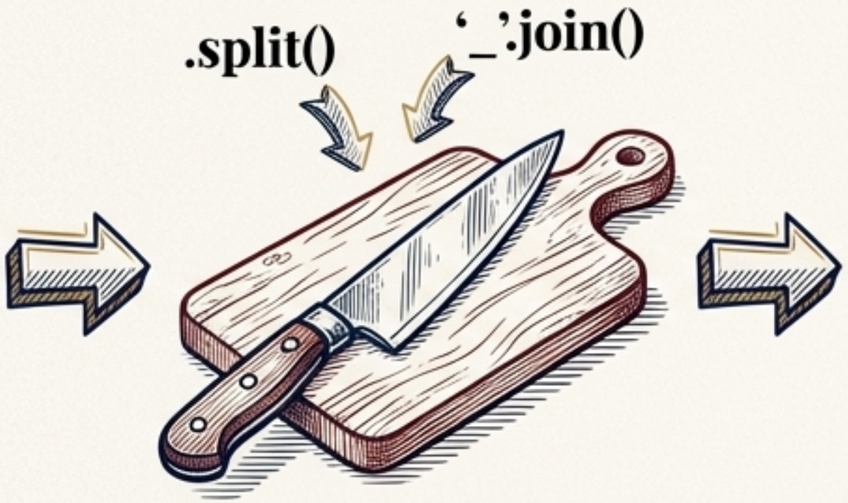
✓ `1` = The passenger survived.

✗ `0` = The passenger did not survive.

Feature Engineering is Like Preparing a Meal

A model can't easily understand a messy string of text and numbers. We must first **"prepare" our data by extracting the useful parts into new, clean features.** This process is called **Feature Engineering.**

Raw Ticket



Preparation Tools

Ticket_item: STON/O2.

Ticket_number: 3101282

Clean Features

The original messy ticket

```
# The original messy ticket
df["Ticket"]

# We apply a function to split the ticket...
df["Ticket_item"] =
df["Ticket"].apply(ticket_item)

# ...and another to isolate the number.
df["Ticket_number"] =
df["Ticket"].apply(ticket_number)
```

We apply a function to the ticket...

...and another to isolate the number.

Anatomy of a Python Function: The `ticket_item` Tool

The Python Function

```
def ticket_item(x):
    items = x.split(" ")    # 1. Splits text at spaces into a list.

    if len(items) == 1:     # 2. Checks if the list has only one item.
        return "NONE"       # 3. If so, assigns "NONE".

    return "_".join(items[0:-1])  # 4. If not, rejoins all but the last item.
```

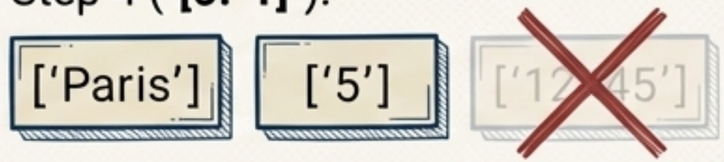
Step-by-Step Execution

Input: Ticket is `Paris 5 12345`

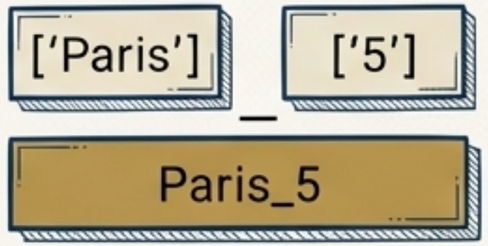
Step 1 (`**.split**`):

Length = 3. `**if**` is false.

Step 4 (`**[0:-1]**`):

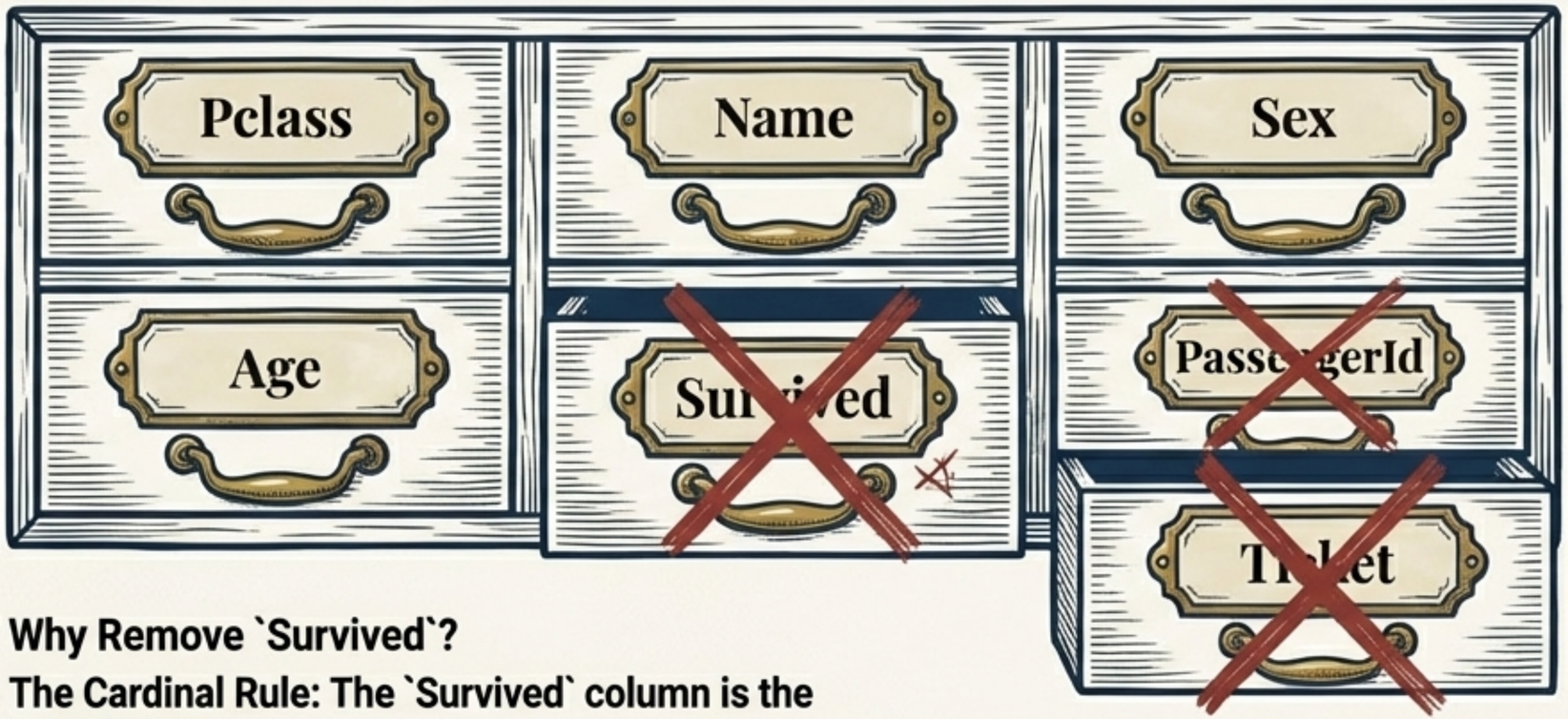


Step 4 (`**.join**`):



Key Takeaway: By creating unique tokens like **`Paris_5`**, we turn a city and a number into a single, meaningful category the model can learn from, just like the difference between 'Blue' and 'Berry' vs. 'Blueberry'.

To Learn, the Model Must Not Cheat



Why Remove `PassengerId`?

It's just a random number. Learning that 'Passenger #4 survived" teaches the model a useless, coincidental rule.

This is random noise.

Why Remove `Survived`?

The Cardinal Rule: The `Survived` column is the Answer Key.

If we show the model the answers while it's 'studying,' it will a **perfect 100% score by simply copying the answer.** It wouldn't learn any real patterns, making it useless for predicting passengers it has never seen before.

```
input_features.remove("Survived")      # Hide the answer key.
input_features.remove("PassengerId")  # Ignore random noise.
input_features.remove("Ticket")     # Use our clean features instead.
```

Accuracy: 82.6%

The First Trial: Training a Baseline Model

With our data prepared and our features selected, it's time to train our first model. The `model.fit()` **command is the "study session"** where the AI analyzes the training data to find patterns that correlate with survival.

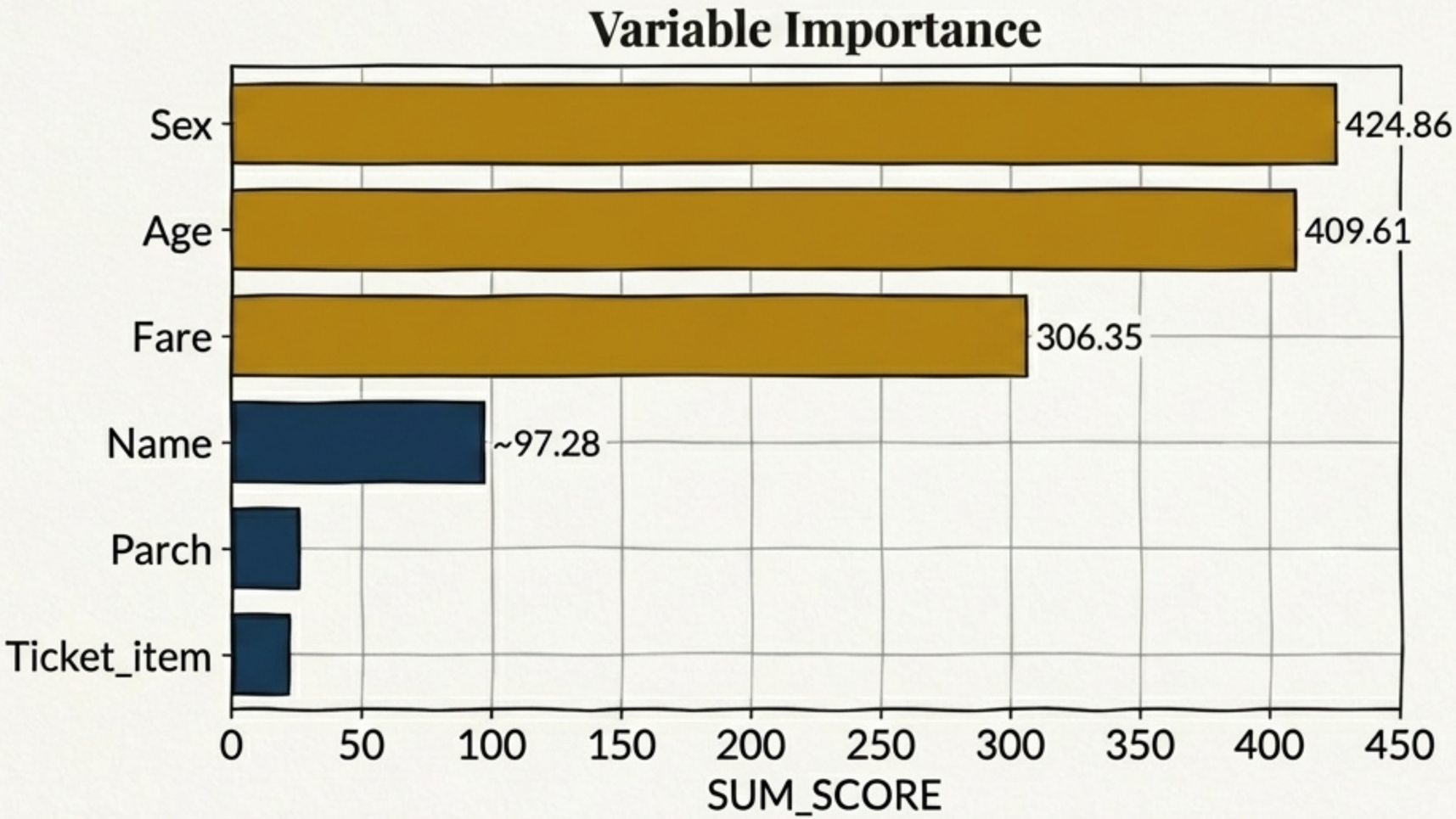
```
# Define the model with default settings
model = tfdf.keras.GradientBoostedTreesModel(
    features=[tfdf.keras.FeatureUsage(name=n) for n in input_features],
    random_seed=1234
)

# Train the model on our dataset
model.fit(train_ds)

# Check its performance
self_evaluation = model.make_inspector().evaluation()
```

Accuracy: 0.8260869...
Loss: 0.8608942...

The Model's Wisdom: What Clues Matter Most?

Accuracy: 82.6%

The model confirms our intuition. The most powerful predictors of survival were a passenger's **Sex**, **Age**, and **Fare**.



`model.summary()` lets us peek inside the trained model.

The 'Variable Importance' score shows which features the model used most often to make its decisions.

The 'women and children first' protocol is not just a historical fact; it's a clear mathematical pattern the model discovered on its own.

The Peril of Overfitting

Accuracy: 76.0%

To improve our score, we tried a more complex model with manually tuned 'hyperparameters'—like settings on a camera. We increased the number of trees and allowed for highly specific rules.

```
model = tfdf.keras.GradientBoostedTreesModel(
    num_trees=2000,  # More trees
    min_examples=1,  # Highly specific rules
    shrinkage=0.05,
    ...
)
```

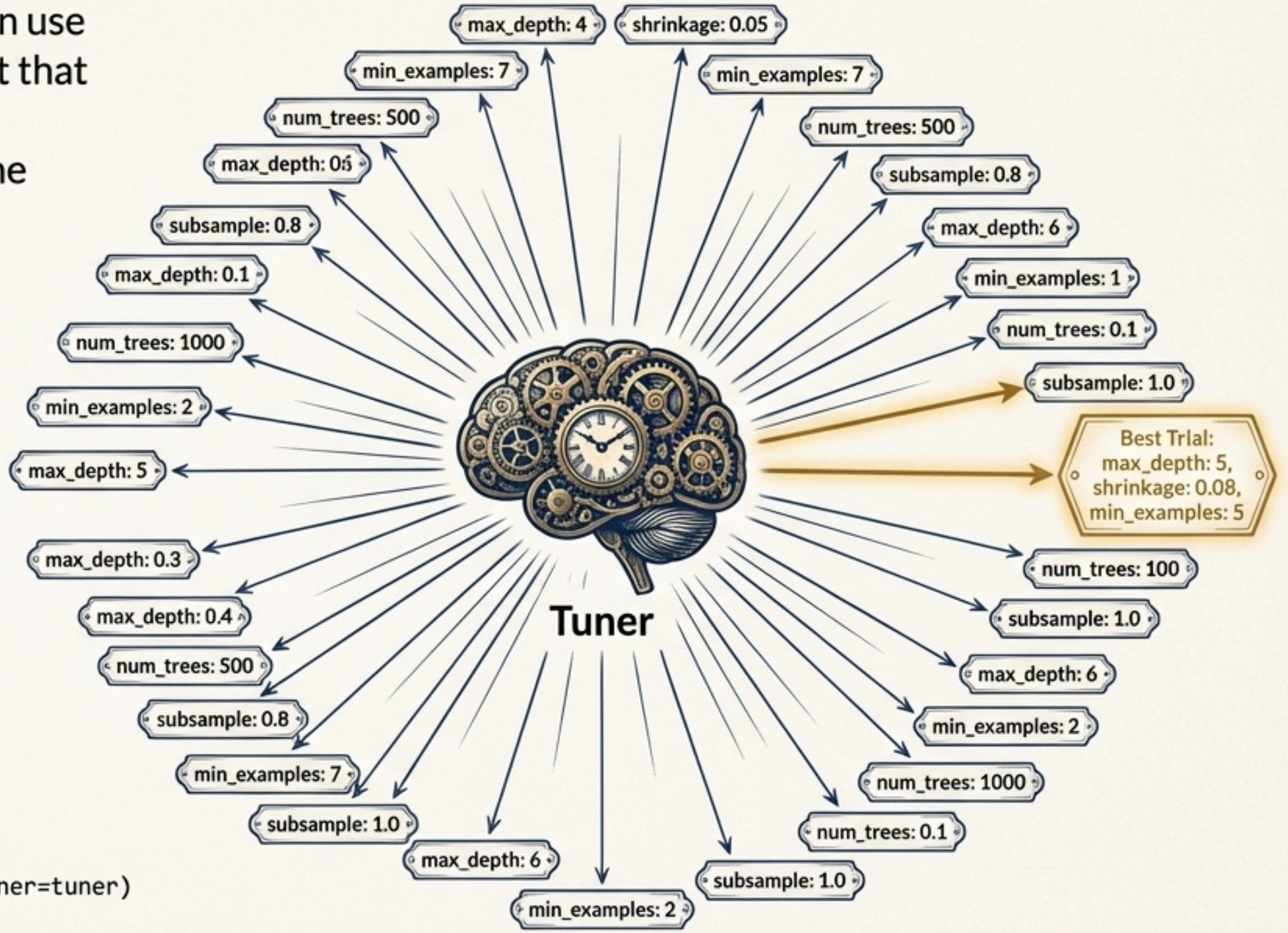
Accuracy: 0.760869...
Loss: 1.015421...

Our model 'memorized' the training data instead of learning general rules. This is **overfitting**, a common trap in machine learning.

A Smarter Approach: Automated Tuning

Accuracy: 76.0%

Instead of guessing the best settings, we can use a `Tuner`. This is like a professional assistant that automatically tests thousands of different combinations of hyperparameters to find the optimal recipe for our model.



The Code Setup

```
# Define the search space for the tuner
tuner = tfdf.tuner.RandomSearch(num_trials=1000)
tuner.choice("min_examples", [2, 5, 7, 10])
tuner.choice("max_depth", [3, 4, 5, 6, 8])
# ... more settings ...

# Pass the tuner to the model
tuned_model = tfdf.keras.GradientBoostedTreesModel(tuner=tuner)
tuned_model.fit(train_ds)
```

The Triumph: A 91.8% Accurate Model

After testing 1,000 different configurations, the tuner found the winning combination. The result is a significant improvement in both accuracy and confidence (lower loss).

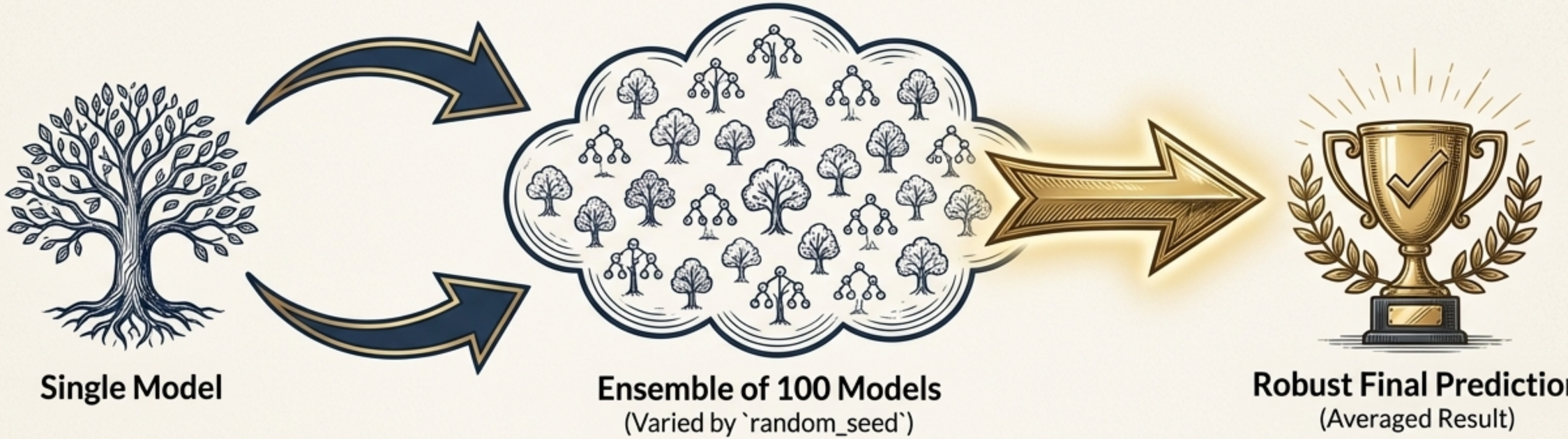
Accuracy: 0.917808...
Loss: 0.650358...

Insight: The top trials often used a `max_depth` of only 3 or 4.

Takeaway: The most accurate model wasn't the most complex. The tuner discovered that shallow, robust trees performed better than the deep, overfitted ones we built manually.

The Final Technique: The Wisdom of the Crowd

To make our prediction even more robust, we can build an **Ensemble**. Instead of relying on a single model, we train 100 different models, each with a unique `random_seed`. We then average their predictions together.



Single Model

Ensemble of 100 Models
(Varied by `random_seed`)

Robust Final Prediction
(Averaged Result)

Key Concept: The `honest=True` Parameter

When training our ensemble, we use an **`honest`** setting. This forces the model to use **different sets** of data to build the tree structure vs. calculating the final predictions in the leaves. This is a **regularization technique** that **reduces bias** and helps the model generalize better to unseen data.

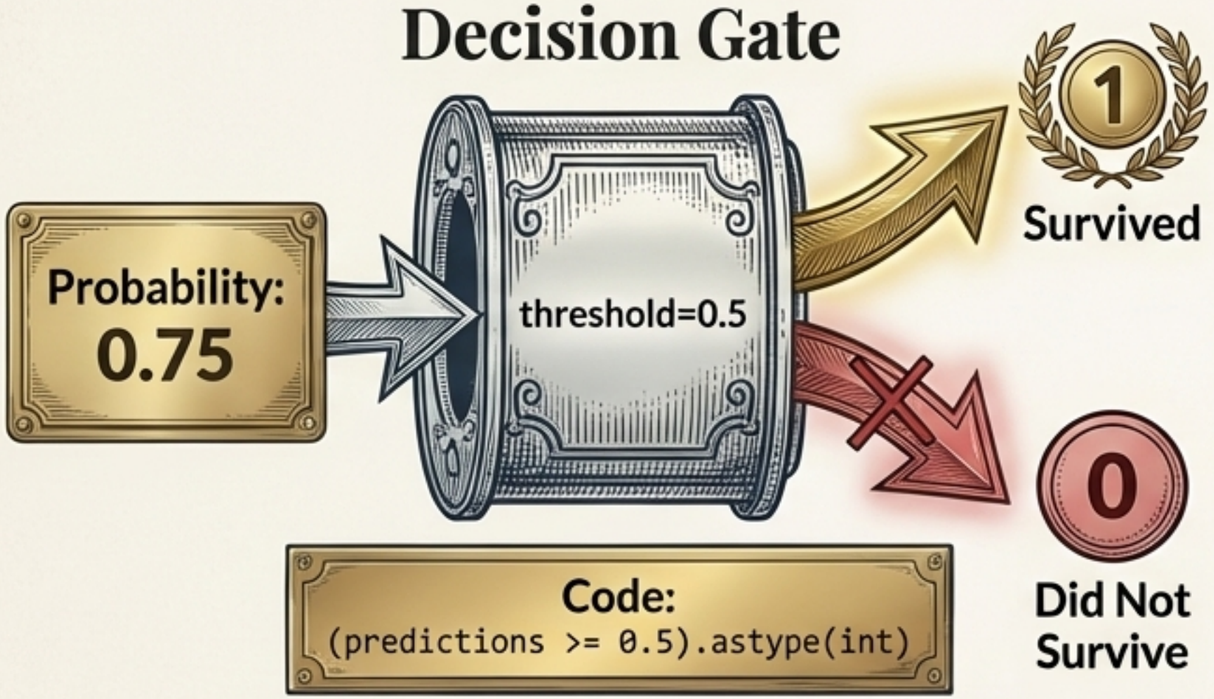
The Logic

If one model makes a mistake due to a random fluke, the other **99 act as a safety net**, pulling the final prediction back towards the true pattern.

Forging the Final Submission

The last step is to use our trained ensemble to predict survival for the `serving_df` (the test set) and format the results for Kaggle.

Decision Gate



If probability is >= 0.5, **`Survived` = 1.**

If probability is < 0.5, **`Survived` = 0.**

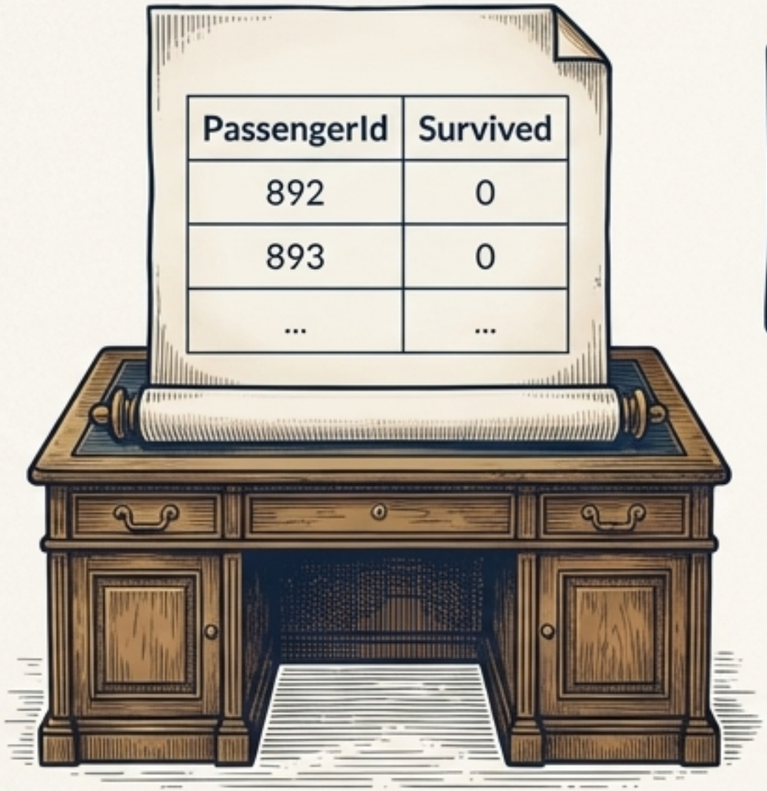
Final Verification

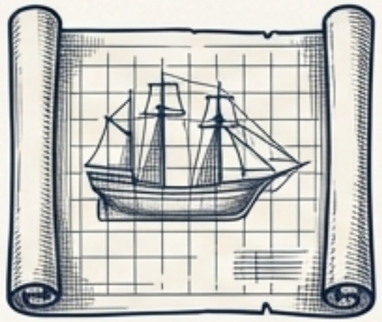
Our Ensemble's Prediction:
143 Survivors / 418 Total Passengers
= 34.2% Survival Rate.

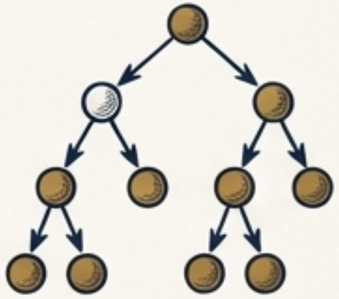
Historical Fact:
The actual survival rate on the Titanic was **~32%.**

Key Insight: Our model's prediction rate is remarkably close to the historical reality, giving us high confidence in our results.

The Quest is Complete







- **Engineer Features**: Clean and transform data to reveal hidden patterns.
- **Deconstruct Code**: Understand the logic behind the tools we use.
- **Battle Overfitting**: Recognize and defeat a common machine learning challenge.
- **Master Advanced Techniques:** Use automated tuning and ensembling to achieve state-of-the-art results.

The true prize isn't just the accuracy score; it's the mastery of the process and the ability to find a clear story within complex data.