



AI as a Service

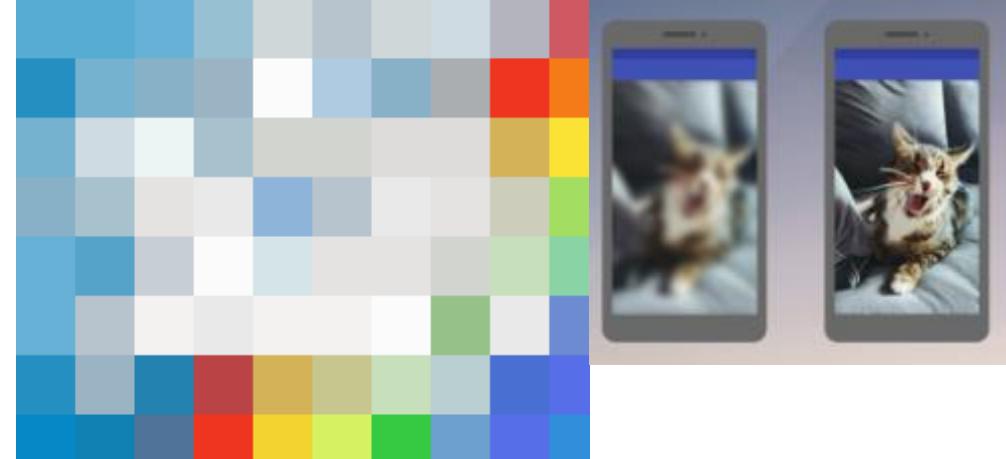
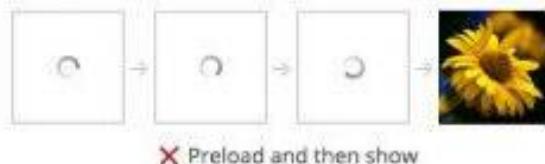
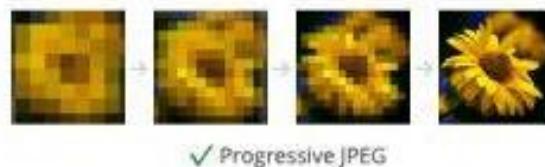
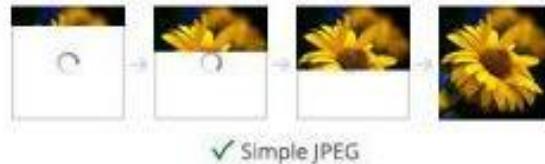


Table of Content

1. Classification Algorithms: 3
2. Evaluation metrics for classification: 93
3. Creating a Model Service: 181
4. Neural networks and deep learning: 202
5. Serving models with Kubernetes and Kubeflow: 227
6. Serverless Deep Learning: 313

1. Classification Algorithms

Why This is Different



ADEPT Method for Learning	
Analogy	Tell me what it's like.
Diagram	Help me visualize it.
Example	Allow me to experience it.
Plain English	Describe it with everyday words.
Technical Definition	Discuss the formal details.

What you will Learn

- Doing exploratory data analysis for identifying important features
- Encoding categorical variables to use them in machine learning models
- Using logistic regression for classification

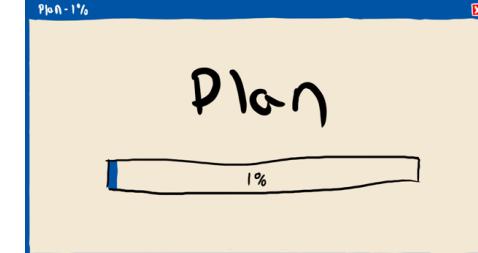


Churn Project

Are customers going to leave the company?



The Plan for the Project



1. First, we download the dataset and do some initial preparation: rename columns and change values inside columns to be consistent throughout the entire dataset.
2. Then we split the data into train, validation, and test so we can validate our models.
3. As part of the initial data analysis, we look at feature importance to identify which features are important in our data.
4. We transform categorical variables into numeric so we can use them in the model.
5. Finally, we train a logistic regression model.



The Dataset

- **Services of the customers** — phone; multiple lines; internet; tech support and extra services such as online security, backup, device protection, and TV streaming
- **Account information** — how long they have been clients, type of contract, type of payment method
- **Charges** — how much the client was charged in the past month and in total
- **Demographic information** — gender, age, and whether they have dependents or a partner
- **Churn** — yes/no, whether the customer left the company within the past month

<https://www.kaggle.com/blastchar/telco-customer-churn>

Initial data preparation

Do your Imports

```
import pandas as pd
```

```
import numpy as np
```

```
import seaborn as sns
```

```
from matplotlib import pyplot as plt
```

```
%matplotlib inline
```



Read the Dataset

```
df = pd.read_csv('WA_Fn-UseC_-Telco-Customer-Churn.csv')
```

```
len(df)
```

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	...
0	7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	...
1	5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	...
2	3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	...
3	7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	...
4	9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	...

Transpose the Dataset to Make it Wide (not long)

df.head().T

	0	1	2
customerID	7590-VHVEG	5575-GNVDE	3668-QPYBK
gender	Female	Male	Male
SeniorCitizen	0	0	0
Partner	Yes	No	No
Dependents	No	No	No
tenure	1	34	2
PhoneService	No	Yes	Yes
MultipleLines	No phone service	No	No
InternetService	DSL	DSL	DSL
OnlineSecurity	No	Yes	Yes
OnlineBackup	Yes	No	Yes
DeviceProtection	No	Yes	No
TechSupport	No	No	No
StreamingTV	No	No	No
StreamingMovies	No	No	No
Contract	Month-to-month	One year	Month-to-month
PaperlessBilling	Yes	No	Yes
PaymentMethod	Electronic check	Mailed check	Mailed check
MonthlyCharges	29.85	56.95	53.85
TotalCharges	29.85	1889.5	108.15
Churn	No	No	Yes

Data Types

```
df.dtypes
```

```
total_charges =  
pd.to_numeric(df.TotalCharges,  
errors='coerce')
```

df.dtypes	
customerID	object
gender	object
SeniorCitizen	int64
Partner	object
Dependents	object
tenure	int64
PhoneService	object
MultipleLines	object
InternetService	object
OnlineSecurity	object
OnlineBackup	object
DeviceProtection	object
TechSupport	object
StreamingTV	object
StreamingMovies	object
Contract	object
PaperlessBilling	object
PaymentMethod	object
MonthlyCharges	float64
TotalCharges	object
Churn	object
dtype: object	

SeniorCitizen is integer

TotalCharges is not correctly identified as a numeric type (float or int)

Consider Empties...

```
df[total_charges.isnull()]['customerID', 'TotalCharges']
```

```
df.TotalCharges = pd.to_numeric(df.TotalCharges, errors='coerce')
```

```
df.TotalCharges = df.TotalCharges.fillna(0)
```

Column Names and Naming Conventions

```
df.columns = df.columns.str.lower().str.replace(' ', '_')
```

```
string_columns = list(df.dtypes[df.dtypes == 'object'].index)
```

```
for col in string_columns:
```

```
    df[col] = df[col].str.lower().str.replace(' ', '_')
```

Encode Churn Target Variable

```
df.churn = (df.churn == 'yes').astype(int)
```

Split the Data for Testing and Training

```
from sklearn.model_selection import train_test_split
```

```
df_train_full, df_test = train_test_split(df, test_size=0.2, random_state=1)
```

Shuffle			Train			Test		
	customerid	gender	customerid	gender	tenure	customerid	gender	tenure
0	7590-vhveg	female	1	8	7892-pookp	female	28	
1	5575-gnvde	male	34	2	3668-qpybk	male	2	
2	3668-qpybk	male	2	5	9305-cdskc	female	8	
3	7795-cfocw	male	45	6	1452-kiovk	male	22	
4	9237-hqjtu	female	2	3	7795-cfocw	male	45	
5	9305-cdskc	female	8	1	5575-gnvde	male	34	
6	1452-kiovk	male	22	0	7590-vhveg	female	1	
7	6713-okomc	female	10	7	6713-okomc	female	10	
8	7892-pookp	female	28	4	9237-hqjtu	female	2	
9	6388-tabgu	male	62	9	6388-tabgu	male	62	

Train, Test, Validate

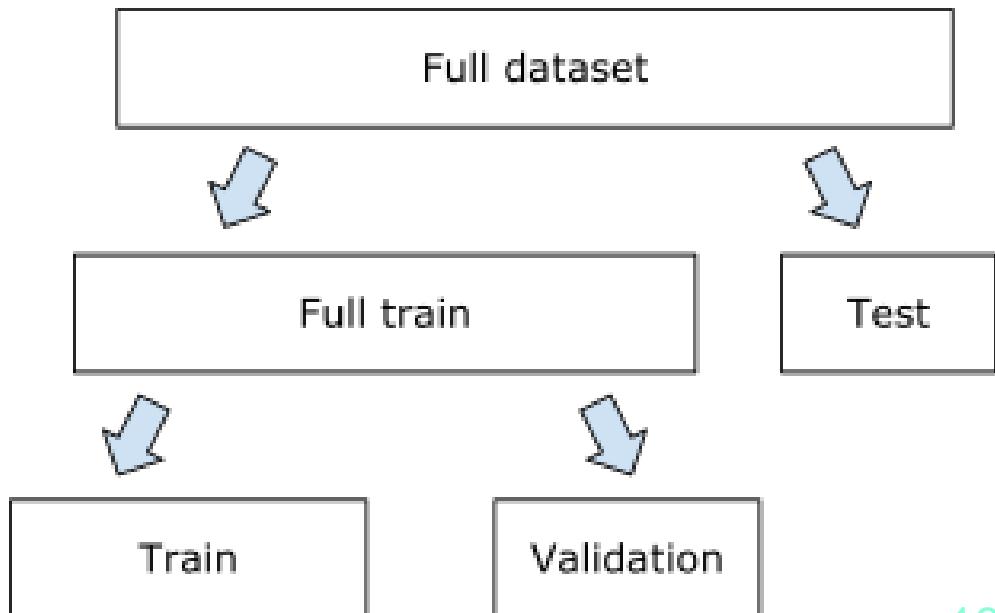
```
df_train, df_val = train_test_split(df_train_full, test_size=0.33, random_state=11)
```

```
y_train = df_train.churn.values
```

```
y_val = df_val.churn.values
```

```
del df_train['churn']
```

```
del df_val['churn']
```



Exploratory Data Analysis

Validate There are No Missing Values

```
df_train_full.isnull().sum()
```

```
df_train_full.isnull().sum()
```

customerid	0
gender	0
seniorcitizen	0
partner	0
dependents	0
tenure	0
phoneservice	0
multiplelines	0
internetservice	0
onlinesecurity	0
onlinebackup	0
deviceprotection	0
techsupport	0
streamingtv	0
streamingmovies	0
contract	0
paperlessbilling	0
paymentmethod	0
monthlycharges	0
totalcharges	0
churn	0
dtype:	int64

Validate the Distribution of the Target Variable

```
df_train_full.churn.value_counts()
```

What percentage of customers STOPPED using the services?

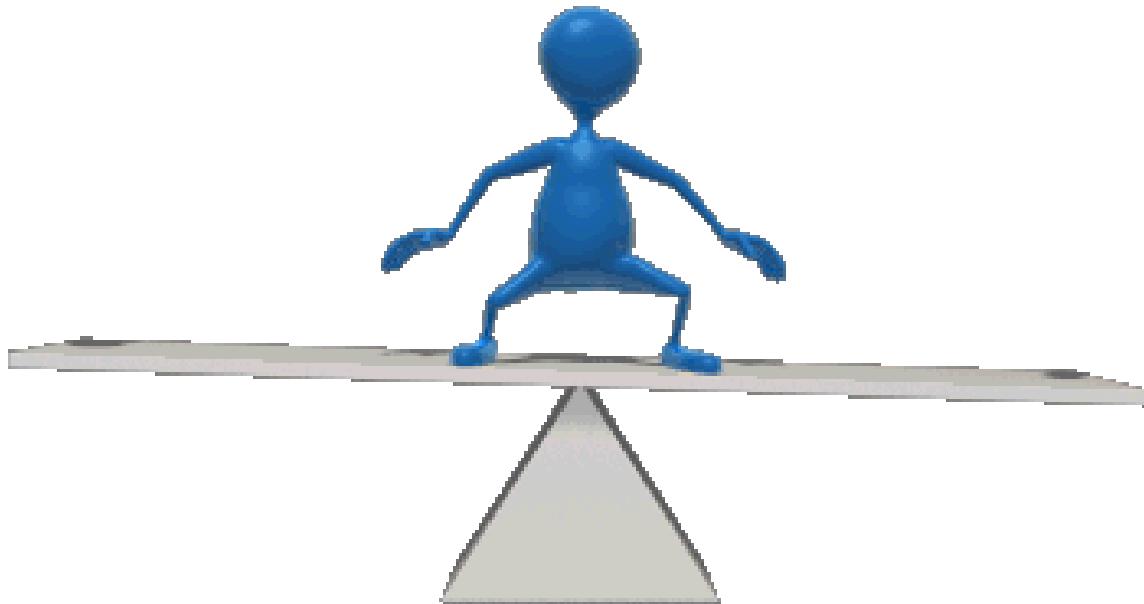
Compute the MEAN of the Target Variable

```
global_mean = df_train_full.churn.mean()
```

```
global_mean = df_train_full.churn.mean()  
round(global_mean, 3)
```

0.27

Imbalanced DAataset



Categorical & Numerical Columns Require Different Treatments

- **categorical**, which will contain the names of categorical variables,
- **numerical**, will have the names of numerical variables

```
categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',  
               'phoneservice', 'multiplelines', 'internetservice',  
               'onlinesecurity', 'onlinebackup', 'deviceprotection',  
               'techsupport', 'streamingtv', 'streamingmovies',  
               'contract', 'paperlessbilling', 'paymentmethod']  
  
numerical = ['tenure', 'monthlycharges', 'totalcharges']
```

Categorical Data

```
df_train_full[categorical].nunique()
```

```
gender              2
seniorcitizen       2
partner             2
dependents          2
phoneservice         2
multiplelines        3
internetservice      3
onlinesecurity       3
onlinebackup          3
deviceprotection     3
techsupport           3
streamingtv          3
streamingmovies       3
contract              3
paperlessbilling      2
paymentmethod          4
dtype: int64
```

Numerical Data

Get the Descriptive statistics for each column (Univariate Analysis)

```
df_train_full[numerical].describe()
```

Correlations

df_train_full.corr()

Feature Importance

Feature Importance

- Knowing how other variables affect the target variable, churn, is the key to understanding the data and building a good model.
 - This process is called **feature importance** analysis
- We have two different kinds of features: categorical and numerical.
 - Each kind has different ways of measuring feature importance, so we will look at each separately.

Feature Importance: Categorical Variables

	customerid	gender	churn
0	7590-vhveg	female	0
1	5575-gnvde	male	0
2	3668-qpybk	male	1
3	7795-cfocw	male	0
4	9237-hqitu	female	1
5	9305-cdskc	female	1
6	1452-kiovk	male	0
7	6713-okomc	female	0
8	7892-pookp	female	1
9	6388-tabgu	male	0

gender == "female"

	customerid	gender	churn
0	7590-vhveg	female	0
4	9237-hqitu	female	1
5	9305-cdskc	female	1
7	6713-okomc	female	0
8	7892-pookp	female	1



	customerid	gender	churn
1	5575-gnvde	male	0
2	3668-qpybk	male	1
3	7795-cfocw	male	0
6	1452-kiovk	male	0
9	6388-tabgu	male	0



gender == "male"

Strategy for Categorical Feature Importance

- So we can look at all the distinct values of a variable.
 - for each variable, there's a group of customers: all the customers who have this value.
 - For each such group, we can compute the churn rate, which will be the group churn rate.
 - When we have it, we can compare it with the global churn rate — churn rate calculated for all the observations at once.
- If the difference between the rates is small, the value is not important when predicting churn because this group of customers is not really different from the rest of the customers.
- On the other hand, if the difference is not small, something inside that group sets it apart from the rest.

An Analyst or machine learning algorithm should be able to pick this up and use it when making predictions.

Feature Importance Based on Gender

Let's check first for the gender variable. To compute the churn rate for all female customers, we first select only rows that correspond to gender == 'female' and then compute the churn rate for them:

```
female_mean = df_train_full[df_train_full.gender == 'female'].churn.mean()
```

```
male_mean = df_train_full[df_train_full.gender == 'male'].churn.mean()
```

Feature Importance based on Partner

```
partner_yes = df_train_full[df_train_full.partner == 'yes'].churn.mean()
```

```
partner_no = df_train_full[df_train_full.partner == 'no'].churn.mean()
```

```
partner_yes = df_train_full[df_train_full.partner == 'yes'].churn.mean()  
print('partner == yes:', round(partner_yes, 3))
```

```
partner_no = df_train_full[df_train_full.partner == 'no'].churn.mean()  
print('partner == no :', round(partner_no, 3))
```

```
partner == yes: 0.205  
partner == no : 0.33
```

Risk Ratio

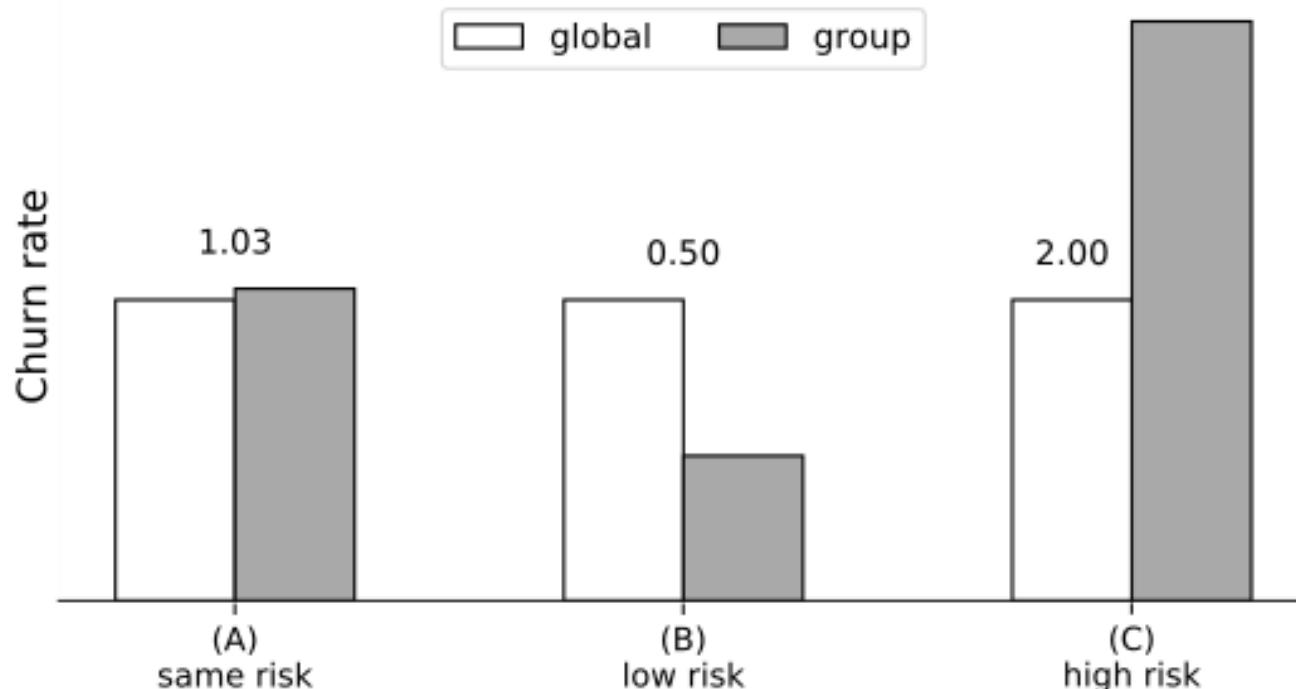
In addition to looking at the difference between the group rate and the global rate, it's interesting to look at the ratio between them. In statistics, the ratio between probabilities in different groups is called risk ratio, where risk refers to the risk of having the effect.

risk = group rate / global rate

For "gender == female", for example, the risk of churning is 1.02:

risk = 27.7% / 27% = 1.02

Risk Ratio



Context for Risk Ratio

- The term risk originally comes from controlled trials, in which one group of patients is given a treatment (a medicine) and the other group isn't (only a placebo).
- Then we compare how effective the medicine is by calculating the rate of negative outcomes in each group and then calculating the ratio between the rates:

risk = negative outcome rate in group 1 / negative outcome rate in group 2

Risk Table

Variable	Value	Churn rate	Risk
Gender	Female	27.7%	1.02
	Male	26.3%	0.97
Partner	Yes	20.5%	0.75
	No	33%	1.22

SQL versus Pandas (Compute Risk RAtio)

SELECT

gender, AVG(churn),

AVG(churn) - global_churn,

AVG(churn) / global_churn

FROM

data

GROUP BY

gender

```
global_mean = df_train_full.churn.mean()
```

```
df_group =
```

```
df_train_full.groupby(by='gender').churn.agg(['mean'])
```

```
df_group['diff'] = df_group['mean'] - global_mean
```

```
df_group['risk'] = df_group['mean'] / global_mean
```

```
df_group
```

gender	mean	diff	risk
female	0.276824	0.006856	1.025396
male	0.263214	-0.006755	0.974980

Risk Ratio for ALL Categorical Variables

```
from IPython.display import display  
  
for col in categorical:  
  
    df_group = df_train_full.groupby(by=col).churn.agg(['mean'])  
  
    df_group['diff'] = df_group['mean'] - global_mean  
  
    df_group['rate'] = df_group['mean'] / global_mean  
  
    display(df_group)
```

Analysis based on Risk Ratio

gender	mean	diff	risk
female	0.276824	0.006856	1.025396
male	0.263214	-0.006755	0.974980

(A) Churn ratio and risk: gender

seniorcitizen	mean	diff	risk
0	0.242270	-0.027698	0.897403
1	0.413377	0.143409	1.531208

(B) Churn ratio and risk: senior citizen

partner	mean	diff	risk
no	0.329809	0.059841	1.221659
yes	0.205033	-0.064935	0.759472

(C) Churn ratio and risk: partner

phoneservice	mean	diff	risk
no	0.241316	-0.028652	0.893870
yes	0.273049	0.003081	1.011412

(D) Churn ratio and risk: phone service

- For gender, there is not much difference between females and males. Both means are approximately the same, and for both groups the risks are close to 1.
- Senior citizens tend to churn more than nonseniors: the risk of churning is 1.53 for seniors and 0.89 for nonseniors.
- People with a partner churn less than people with no partner. The risks are 0.75 and 1.22, respectively.
- People who use phone service are not at risk of churning: the risk is close to 1, and there's almost no difference with the global churn rate. People who don't use phone service are even less likely to churn: the risk is below 1, and the difference with the global churn rate is negative.

Churn Analysis

	mean	diff	risk
techsupport			
no	0.418914	0.148946	1.551717
no_internet_service	0.077805	-0.192163	0.288201
yes	0.159926	-0.110042	0.592390

	mean	diff	risk
contract			
month-to-month	0.431701	0.161733	1.599082
one_year	0.120573	-0.149395	0.446621
two_year	0.028274	-0.241694	0.104730

(A) Churn ratio and risk: tech support

(B) Churn ratio and risk: contract

- Clients with no tech support tend to churn more than those who do.
- People with monthly contracts cancel the contract a lot more often than others, and people with two-year contacts churn very rarely.

Mutual Information : Categorical

“Customers with month-to-month contracts tend to churn a lot more than customers with other kinds of contracts. This is exactly the kind of relationship we want to find in our data. Without such relationships in data, machine learning models will not work — they will not be able to make predictions. The higher the degree of dependency, the more useful a feature is.”

Mutual Information : Categorical

```
from sklearn.metrics import mutual_info_score

def calculate_mi(series):
    return mutual_info_score(series, df_train_full.churn)

df_mi = df_train_full[categorical].apply(calculate_mi)
df_mi = df_mi.sort_values(ascending=False).to_frame(name='MI')

df_mi
```

Mutual Information : Categorical

	MI
contract	0.098320
onlinesecurity	0.063085
techsupport	0.061032
internetservice	0.055868
onlinebackup	0.046923

(A) The most useful features according to the mutual information score.

	MI
partner	0.009968
seniorcitizen	0.009410
multiplelines	0.000857
phoneservice	0.000229
gender	0.000117

(B) The least useful features according to the mutual information score.

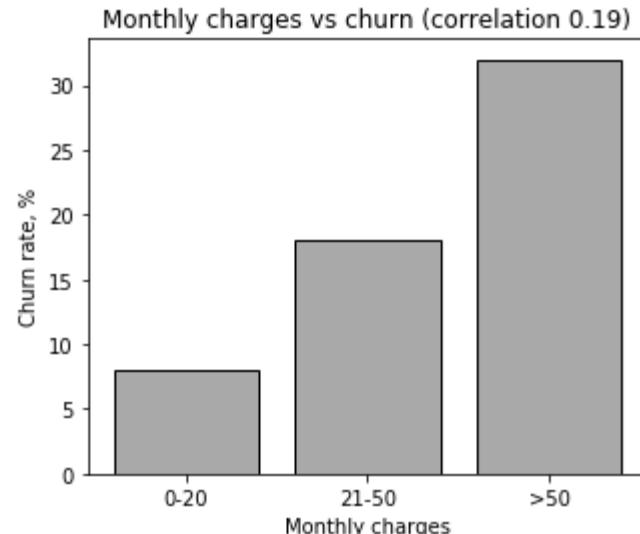
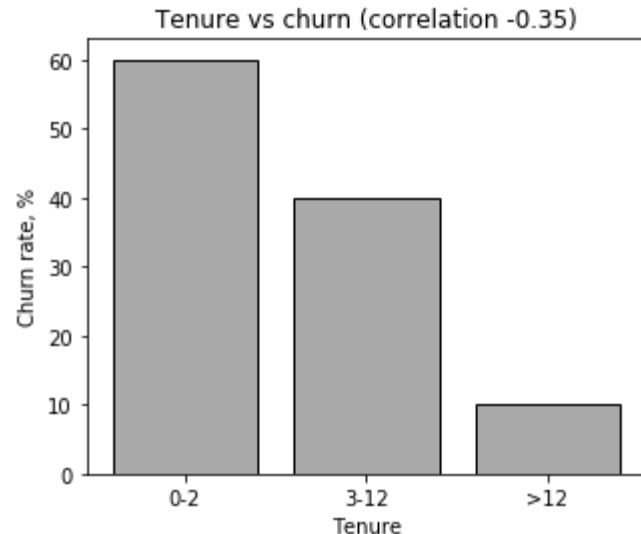
Correlation Coefficient

```
df_train_full[numerical].corrwith(df_train_full.churn)
```

Mutual Information shows the degree of dependency of Categorical Variables to the Target Variable.

Correlation does the same with Numeric Variables.

Correlation



correlation

tenure	-0.351885
monthlycharges	0.196805
totalcharges	-0.196353

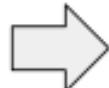
Feature Engineering

Transform all categorical variables
to numeric forms

One Hot Encoding

One Hot Encoding Concept

gender	contract
male	monthly
female	yearly

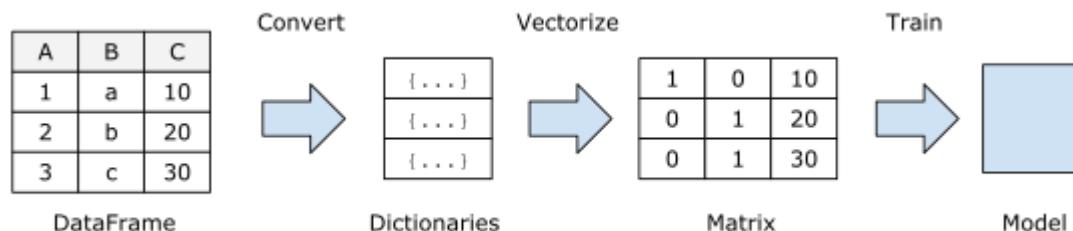


gender		contract		
female	male	monthly	yearly	2-year
0	1	1	0	0
1	0	0	1	0

DictVectorizer

- To use it, we need to convert our dataframe to a list of dictionaries. It's very simple to do in Pandas. Use the `to_dict` method with the `orient="records"` parameter:

```
train_dict = df_train[categorical + numerical].to_dict(orient='records')
```



```
{'gender': 'male',  
 'seniorcitizen': 0,  
 'partner': 'yes',  
 'dependents': 'yes',  
 'phoneservice': 'yes',  
 'multiplelines': 'no',  
 'internetservice': 'no',  
 'onlinesecurity': 'no_internet_service',  
 'onlinebackup': 'no_internet_service',  
 'deviceprotection': 'no_internet_service',  
 'techsupport': 'no_internet_service',  
 'streamingtv': 'no_internet_service',  
 'streamingmovies': 'no_internet_service',  
 'contract': 'two_year',  
 'paperlessbilling': 'no',  
 'paymentmethod': 'mailed_check',  
 'tenure': 12,  
 'monthlycharges': 19.7,  
 'totalcharges': 258.35}
```

Dictionary Vectorizer

```
from sklearn.feature_extraction import DictVectorizer
```

```
dv = DictVectorizer(sparse=False)
```

```
dv.fit(train_dict)
```

```
X_train = dv.transform(train_dict)
```

Peek at the Vectorized Data

```
X_train[0]
```

```
dv.get_feature_names()
```

Exercise

#How would DictVectorizer encode the following list of

#dictionaries:

records = [

{'total_charges': 10, 'paperless_billing': 'yes'},

{'total_charges': 30, 'paperless_billing': 'no'},

{'total_charges': 20, 'paperless_billing': 'no'}

]

Machine learning

ML for Classification

Predictive Analytics from the clean
Telco Dataset

Logistic Regression

Linear Regression

$$g(x_i) = w_0 + x_i^T w$$

where

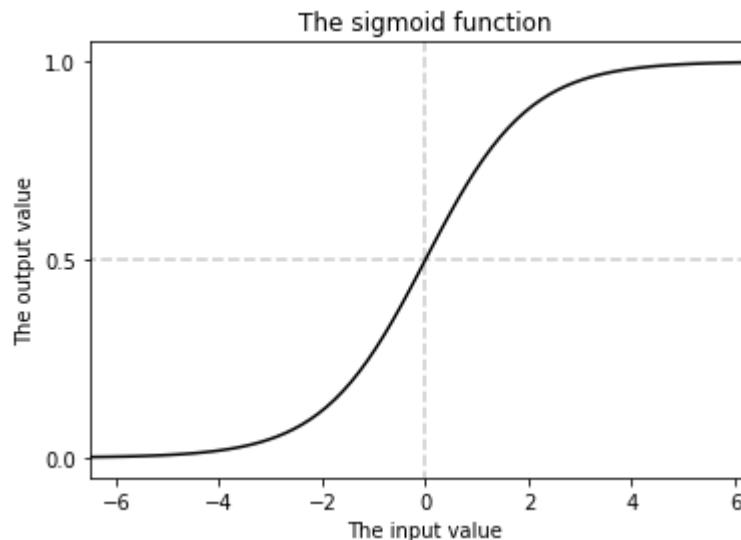
x_i is the feature vector corresponding to the i th observation,

w_0 is the bias term,

w is a vector with the weights of the model.

Logistic Regression

$$g(x_i) = \text{sigmoid}(w_0 + x_i^T w)$$
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$



Linear Regression from Scratch in Python

```
def linear_regression(xi):  
    result = bias  
  
    for j in range(n):  
        result = result + xi[j] * w[j]  
  
    return result
```

Logistic Regression from Scratch using Python

```
import math

def logistic_regression(xi):
    score = bias
    for j in range(n):
        score = score + xi[j] * w[j]
    prob = sigmoid(score)
    return prob

def sigmoid(score):
    return 1 / (1 + math.exp(-score))
```

Exercise

Why do we need the Sigmoid Function for Logistic Regression?

Training the Logistic Regression Model

Training the Model

```
from sklearn.linear_model import LogisticRegression  
  
model = LogisticRegression(solver='liblinear', random_state=1)  
  
model.fit(X_train, y_train)
```

One Hot Encoding

```
val_dict = df_val[categorical + numerical].to_dict(orient='records')
```

```
X_val = dv.transform(val_dict)
```

```
y_pred = model.predict_proba(X_val)
```

Understanding the Predictions

- The predictions of the model: a two-column matrix.
- The first column is the probability that the target=0
- The second column is the probability tha the target=1

```
model.predict_proba(X_val)  
  
array([[0.76508957, 0.23491043],  
       [0.73113584, 0.26886416],  
       [0.68054864, 0.31945136],  
       ...,  
       [0.94274779, 0.05725221],  
       [0.38476995, 0.61523005],  
       [0.9387273 , 0.0612727 ]])
```

Probability that the observation belongs to the negative class: i.e. customer will not churn

Probability that the observation belongs to the positive class, i.e. customer will churn

But We only need ONE column

```
y_pred = model.predict_proba(X_val)[:, 1]
```

```
y_pred >= 0.5
```

y_pred									
0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	
↓ ≥ 0.5 ↓									
False	False	False	False	True	True	True	True	True	

Introducing Accuracy

```
churn = y_pred >= 0.5
```

```
(y_val == churn).mean() #Quality Measure called ACCURACY
```

$y_{\text{val}} == \text{churn}$

Target data: 1 if customer churns, 0 if not

Predictions: True if we think the customer churns, False if we think they won't

Accuracy

y_val	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	0	0	1					
0	1	0	0	1							
==											
churn	<table border="1"><tr><td>False</td><td>True</td><td>True</td><td>False</td><td>True</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	False	True	True	False	True	0	1	1	0	1
False	True	True	False	True							
0	1	1	0	1							
	 Cast to integer										

Accuracy

```
(y_val == churn).mean()
```

{ }

Boolean array is cast to
integer array



Computing its mean

Model Interpretation

Understand What You've Done : Coefficients

w0 is the bias term.

w = (w1, w2, ..., wn) is the weights vector

- We can get the bias term from model.intercept_[0]. When we train our model on all features, the bias term is -0.12
- The rest of the weights are stored in model.coef_[0]

```
dict(zip(dv.get_feature_names(), model.coef_[0].round(3)))
```

Weights

```
{"contract=month-to-month": 0.563,  
'contract=one_year': -0.086,  
'contract=two_year': -0.599,  
'dependents=no': -0.03,  
'dependents=yes': -0.092,  
... # the rest of the weights is omitted  
'tenure': -0.069,  
'totalcharges': 0.0}
```

Prepare a Small Subset to Break Down the Categoricals

```
small_subset = ['contract', 'tenure', 'totalcharges']

train_dict_small = df_train[small_subset].to_dict(orient='records')

dv_small = DictVectorizer(sparse=False)

dv_small.fit(train_dict_small)

X_small_train = dv_small.transform(train_dict_small)

dv_small.get_feature_names()
1 ['contract=month-to-month',
2 'contract=one_year',
3 'contract=two_year',
4 'tenure',
5 'totalcharges']
```

Train the Small Subset

```
model_small = LogisticRegression(solver='liblinear', random_state=1)
```

```
model_small.fit(X_small_train, y_train)
```

```
model_small.intercept_[0] #Check the bias
```

```
dict(zip(dv_small.get_feature_names(), model_small.coef_[0].round(3))) #Check the other weights
```

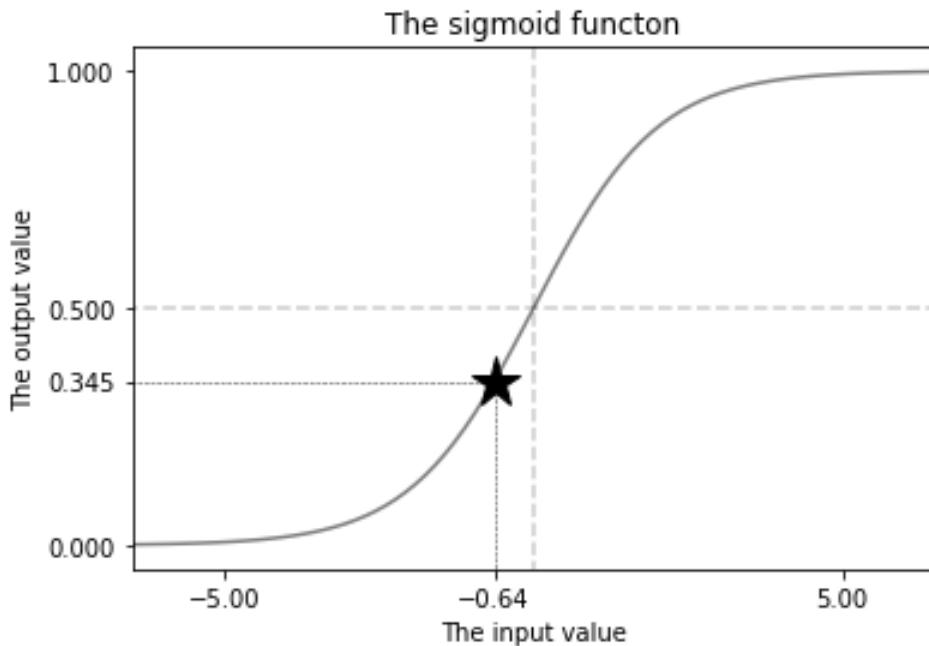
```
1 {'contract=month-to-month': 0.91,
2 'contract=one_year': -0.144,
3 'contract=two_year': -1.404,
4 'tenure': -0.097,
5 'totalcharges': 0.000}
```

The Big Picture with the Categoricals

bias	contract			tenure	charges
	month	year	2-year		
w0	w1	w2	w3	w4	w5
-0.639	0.91	-0.144	-1.404	-0.097	0.0

The Sigmoid Function

The bias term -0.639 on the sigmoid curve. The resulting probability is less than 0.5 so the average customer is more likely not to churn.



Understanding The Importance of Categories

```
dict(zip(dv_small.get_feature_names(), model_small.coef_[0].round(3)))
```

'contract=month-to-month': 0.91,

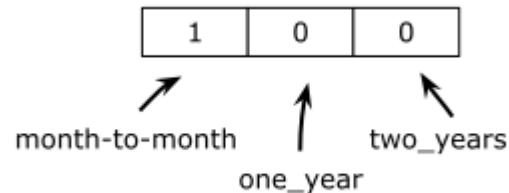
```
1 {'contract=month-to-month': 0.91,  
2 'contract=one_year': -0.144,  
3 'contract=two_year': -1.404,  
4 'tenure': -0.097,  
5 'totalcharges': 0.000}
```

'contract=one_year': -0.144,

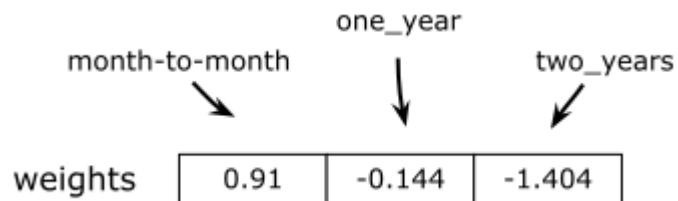
'contract=two_year': -1.404.

Build your Intuition

The one-hot encoded representation for a customer with a month to month contract



The weights of the month to month, 1 year, 2 year contract features



Making a Prediction with One Hot Encoded Categorical Data

- The dot product between the one-hot encoding representation of the contract variable and the corresponding weights.
- The result is 0.91 which is the weight of the hot feature.

$$\begin{array}{c} \text{month-to-month} \quad \text{one_year} \quad \text{two_years} \\ \downarrow \quad \downarrow \quad \downarrow \\ \text{contract} \quad \boxed{1} \quad 0 \quad 0 \\ \times \\ \text{weights} \quad \boxed{0.91} \quad -0.144 \quad -1.404 \\ = \\ \boxed{1} \cdot \boxed{0.91} + \boxed{0} \cdot \boxed{-0.144} + \boxed{0} \cdot \boxed{-1.404} \\ \text{month-to-month} \quad \text{one_year} \quad \text{two_years} \\ = \\ 0.91 \end{array}$$

Same Thing but a 2 year contract client

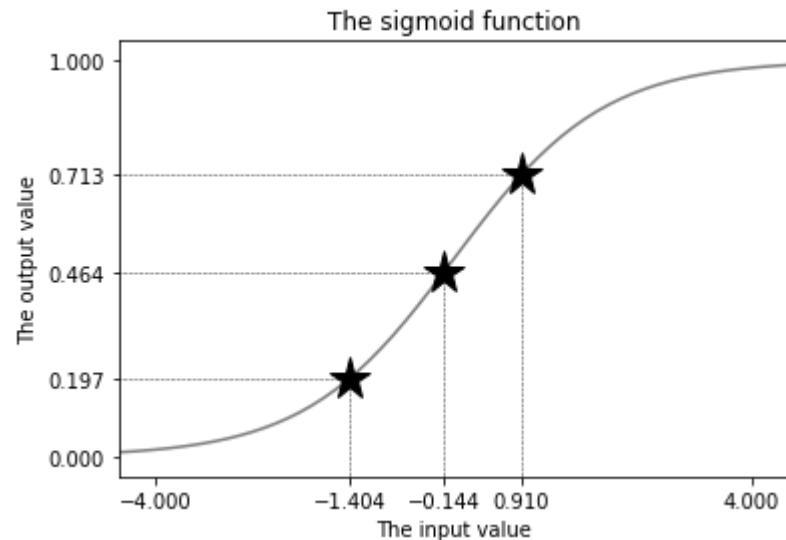
Only the HOT feature is ever factored into the model.

$$\begin{array}{c} \text{month-to-month} \quad \text{one_year} \quad \text{two_years} \\ \downarrow \quad \downarrow \quad \downarrow \\ \text{contract} \quad \boxed{0} \quad \boxed{0} \quad \boxed{1} \\ \times \\ \text{weights} \quad \boxed{0.91} \quad \boxed{-0.144} \quad \boxed{-1.404} \\ = \\ \boxed{0} \cdot \boxed{0.91} + \boxed{0} \cdot \boxed{-0.144} + \boxed{1} \cdot \boxed{-1.404} \\ \text{month-to-month} \quad \text{one_year} \quad \text{two_years} \\ = \\ -1.404 \end{array}$$

Does the sign (+/-) have meaning?

The sign of the weight matters. If it is positive - churn. Negative - loyal customer.

month-to-month	one_year	two_years	
	↓	↓	
weights	0.91	-0.144	-1.404



Let's Analyze the Numerical Features

The score model calculate for a customer with a m2m contract, 12 months of tenure and total charges of \$1,000

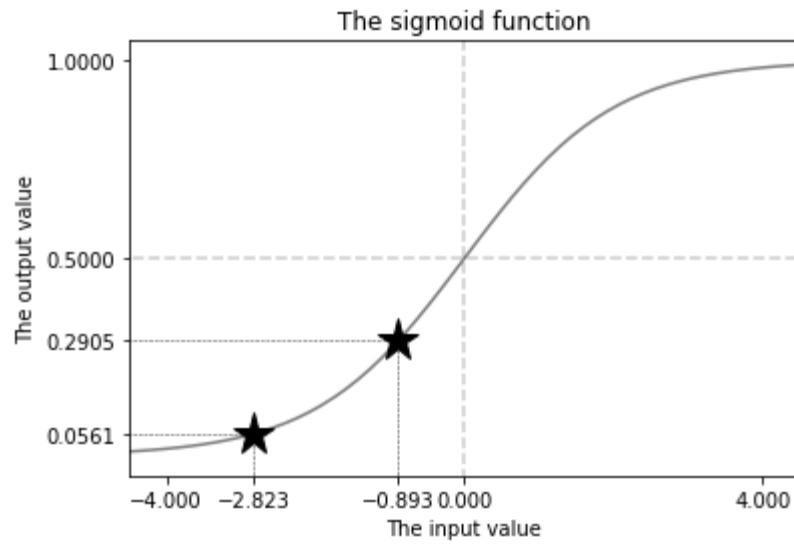
$$-0.639 + 0.91 - 12 \cdot 0.097 + 0 \cdot 1000 = -0.893$$

bias	monthly contract	12 months of tenure	total charges don't matter	negative, so low likelihood of churn
------	------------------	---------------------	----------------------------	--------------------------------------

The score model calculate for a customer with a yearly contract, 24 months of tenure and total charges of \$2,000

$$-0.639 + 0.144 - 24 \cdot 0.097 + 0 \cdot 2000 = -2.823$$

bias	yearly contract	24 months of tenure	total charges don't matter	negative, very low likelihood of churn
------	-----------------	---------------------	----------------------------	--



Using the Churn Model

```
customer = {  
    'customerid': '8879-zkjof',           #Continuation  
    'gender': 'female',                  'deviceprotection': 'yes',  
    'seniorcitizen': 0,                  'techsupport': 'yes',  
    'partner': 'no',                    'streamingtv': 'yes',  
    'dependents': 'no',                 'streamingmovies': 'yes',  
    'tenure': 41,                       'contract': 'one_year',  
    'phoneservice': 'yes',              'paperlessbilling': 'yes',  
    'multiplelines': 'no',              'paymentmethod': 'bank_transfer_(automatic)',  
    'internetservice': 'dsl',           'monthlycharges': 79.85,  
    'onlinesecurity': 'yes',            'totalcharges': 3320.75,  
    'onlinebackup': 'no',               }  
}
```

Vectorized Input

```
X_test = dv.transform([customer])
```

```
#Output
```

```
[[ 0. ,  1. ,  0. ,  1. ,  0. ,  0. ,  0. ,
  1. ,  1. ,  0. ,  1. ,  0. ,  0. ,  0. ,
  1. ,  0. ,  0. ,  1. ,  0. ,  0. ,  0. ,
  0. ,  1. ,  0. ,  1. ,  1. ,  0. ,  1. ,
  0. ,  0. ,  0. ,  0. ,  1. ,  0. ,  0. ,
  0. ,  1. ,  0. ,  0. ,  1. ,  0. ,  0. ,
  1. ,  41. , 3320.75]]
```

Put the Matrix into the Trained Model

```
model.predict_proba(X_test)
```

```
#output
```

```
[[0.93, 0.07]]
```

```
X_test = dv.transform([customer])
model.predict_proba(X_test)[0, 1]
```

```
0.07332577315357781
```

All we need from the matrix is the number at the first row and second column: the probability of churning for this customer. To select this number from the array, we use the brackets operator:

```
model.predict_proba(X_test)[0, 1]
```

Take a look at another customer

```
customer = {  
    'gender': 'female',  
    'seniorcitizen': 1,  
    'partner': 'no',  
    'dependents': 'no',  
    'phoneservice': 'yes',  
    'multiplelines': 'yes',  
    'internetservice': 'fiber_optic',  
    'onlinesecurity': 'no',  
    'onlinebackup': 'no',  
    'deviceprotection': 'no',  
    'techsupport': 'no',  
    'streamingtv': 'yes',  
    'streamingmovies': 'no',  
    'contract': 'month-to-month',  
    'paperlessbilling': 'yes',  
    'paymentmethod': 'electronic_check',  
    'tenure': 1,  
    'monthlycharges': 85.7,  
    'totalcharges': 85.7  
}
```

Let's Make a Prediction

```
X_test = dv.transform([customer])
```

```
model.predict_proba(X_test)[0, 1]
```

```
In [62]: X_test = dv.transform([customer])
model.predict_proba(X_test)[0, 1]
```

```
Out[62]: 0.8321638622459152
```

Project

- Classification models are often used for marketing purposes, and one of the problems it solves is lead scoring.
- A lead is a potential customer who may convert (became an actual customer) or not.
- In this case, the conversion is the target we want to predict.
- You can take a dataset from <https://www.kaggle.com/ashydv/leads-dataset> and build a model for that.
- You may notice that the lead scoring problem is very similar to churn prediction, but in one case we want to get a new client to sign a contract with us, and in another case we want a client not to cancel the contract.

Project 2

- Another popular application of classification is default prediction, which is estimating the risk of a customer's not returning a loan.
- In this case, the variable we want to predict is default, and it also has two outcomes: whether the customer managed to pay back the loan in time (good customer) or not (default).
- There are many datasets online that you can use for training a model, such as <https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients> (or, the same one available via kaggle: <https://www.kaggle.com/pratjain/credit-card-default>).

Summary

Classification

2. Evaluation metrics for classification

Overview

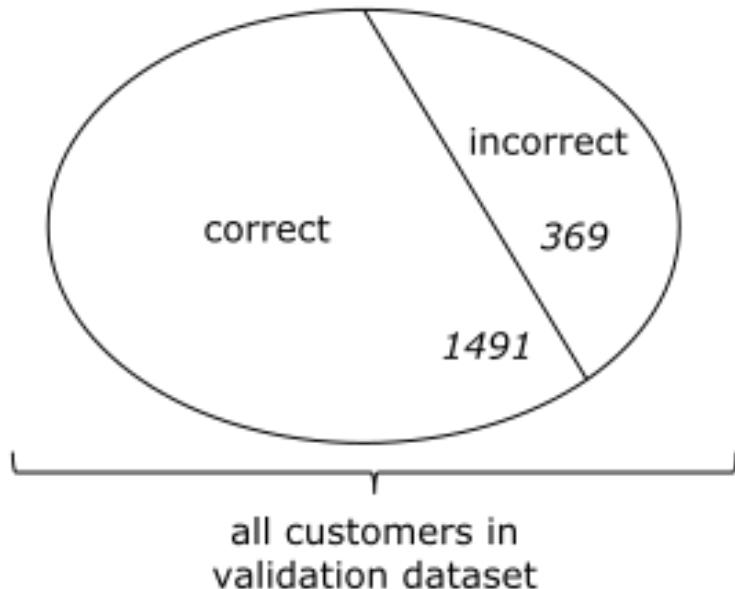
This lesson covers

- Accuracy as a way of evaluating binary classification models and its limitations
- Determining where our model makes mistakes using a confusion table
- Deriving other metrics like precision and recall from the confusion table
- Using ROC (receiver operating characteristics) and AUC (area under the ROC curve) to further understand the performance of a binary classification model

Evaluation metrics

- For this, we use a metric — a function that looks at the predictions the model makes and compares them with the actual values.
- Then, based on the comparison, it calculates how good the model is.
- This is quite useful: we can use it to compare different models and select the one with the best metric value.

Classification accuracy



$$\text{accuracy} = \frac{\text{correct}}{\text{total}} = \frac{1491}{1860} = 80\%$$

Classification accuracy

- Computing accuracy on the validation dataset is easy: we simply calculate the fraction of correct predictions:

```
y_pred = model.predict_proba(X_val)[:, 1] # A  
churn = y_pred >= 0.5 # B  
(churn == y_val).mean() # C
```

Classification accuracy

- Let's open it and add the import statement to import accuracy from Scikit-Learn's metrics package:

```
from sklearn.metrics import accuracy_score
```

Classification accuracy

- Now we can loop over different thresholds and check which one gives the best accuracy:

```
thresholds = np.linspace(0, 1, 11)
```

```
for t in thresholds:
```

```
    churn = y_pred >= t
```

```
    acc = accuracy_score(y_val, churn)
```

```
    print('%0.2f %0.3f' % (t, acc))
```

Classification accuracy

- When we execute the code, it prints the following:

0.00 0.261

0.10 0.595

0.20 0.690

0.30 0.755

0.40 0.782

0.50 0.802

0.60 0.790

0.70 0.774

0.80 0.742

0.90 0.739

1.00 0.739



Classification accuracy

- We first put the values to a list:

```
thresholds = np.linspace(0, 1, 21)
```

```
accuracies = []
```

```
for t in thresholds:
```

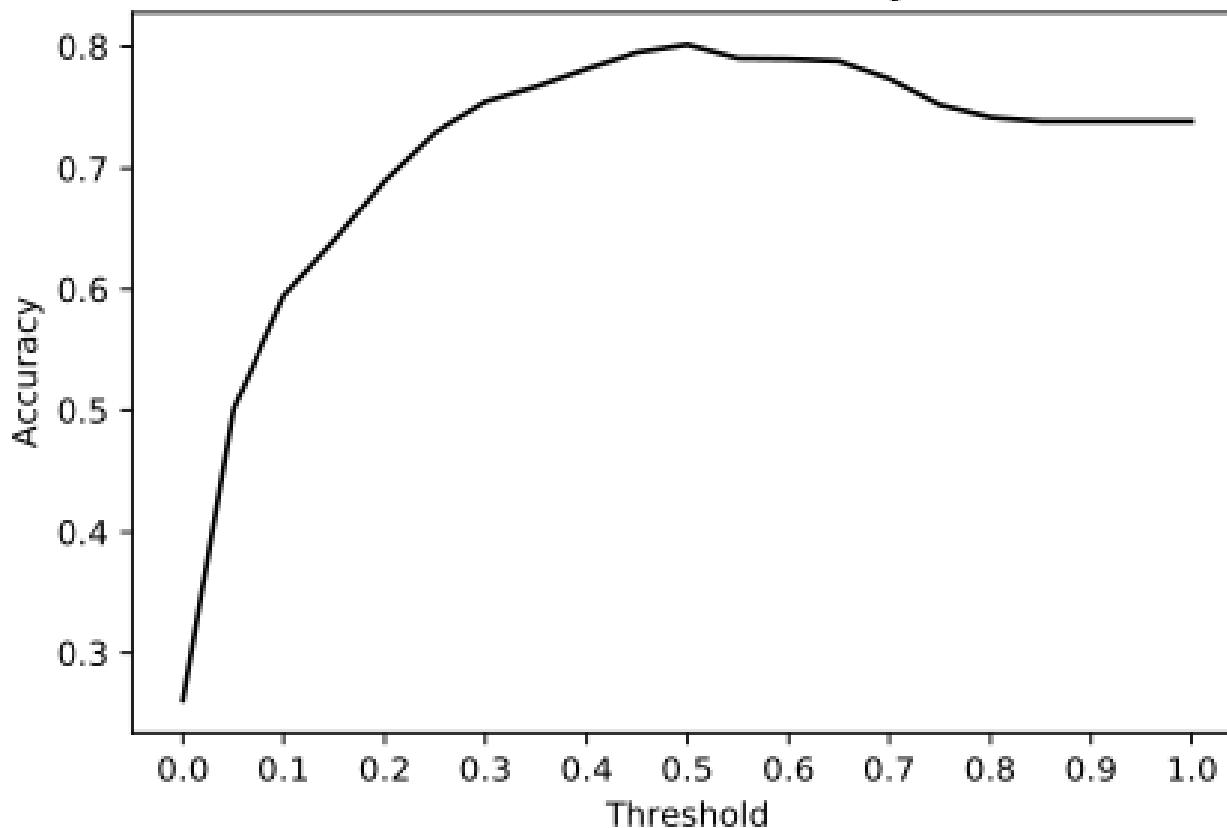
```
    acc = accuracy_score(y_val, y_pred >= t)
```

```
    accuracies.append(acc)
```

- And then we plot these values using Matplotlib:

```
plt.plot(thresholds, accuracies)
```

Threshold vs Accuracy



Classification accuracy

- Let's also check its accuracy, For that, we first make predictions on the validation dataset and then compute the accuracy score:

```
val_dict_small = df_val[small_subset].to_dict(orient='rows')

x_small_val = dv_small.transform(val_dict_small)
y_pred_small = model_small.predict_proba(x_small_val)[:, 1]

churn_small = y_pred_small >= 0.5
accuracy_score(y_val, churn_small)
```

Dummy baseline

- Let's create this baseline prediction:

```
size_val = len(y_val)  
baseline = np.repeat(False, size_val)
```

- To create an array with the baseline predictions we first need to determine how many elements are in the validation set.

Dummy baseline

- Now we can check the accuracy of this baseline prediction using the same code as previously:

```
accuracy_score(baseline, y_val)
```

Dummy baseline

- When we run the code, it shows 0.738.
- This means that the accuracy of the baseline model is around 74%

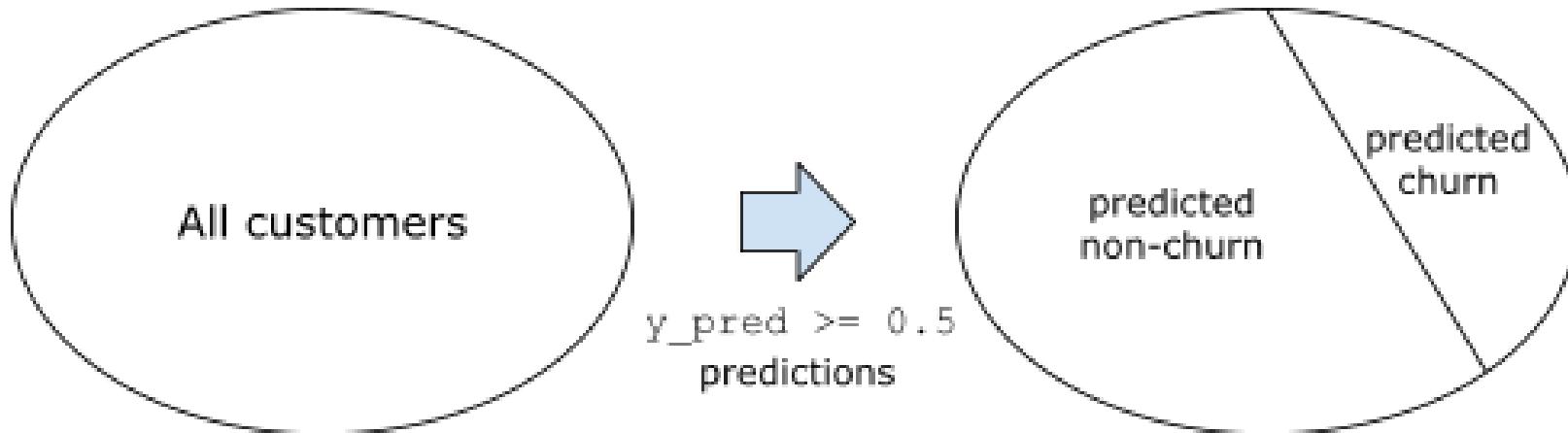
```
size_val = len(y_val)
baseline = np.repeat(False, size_val)
baseline
array([False, False, False, ..., False, False, False])

accuracy_score(baseline, y_val)
0.7387096774193549
```

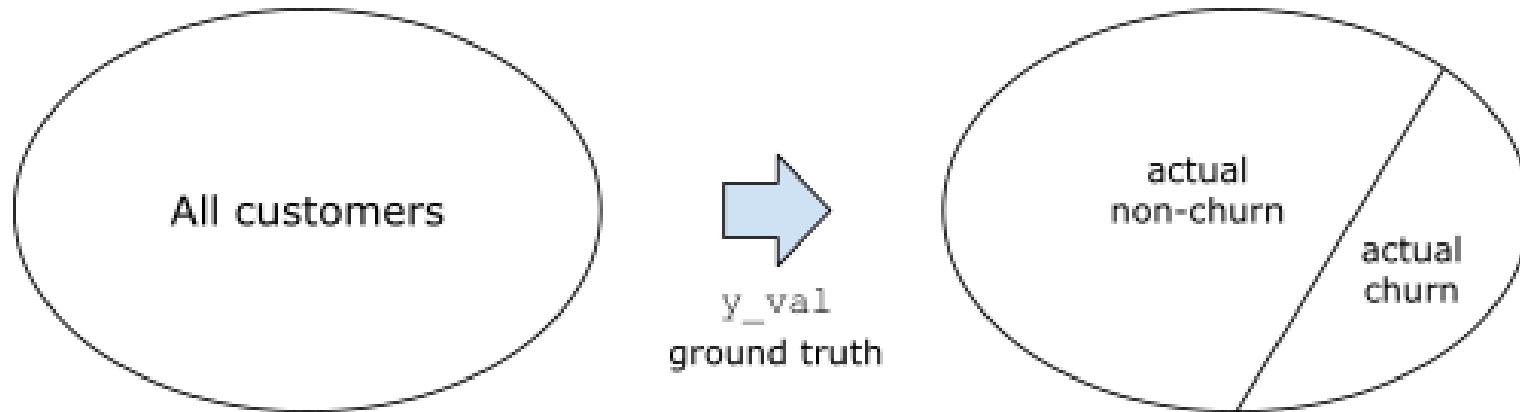
Confusion table

- Even though accuracy is easy to understand, it's not always the best metric.
- What is more, it sometimes can be quite misleading.
- We've already seen it: the accuracy of our model is 80%, and while that seems like a good number, it's just 6% better than the accuracy of a dummy model that always outputs the same prediction of "no churn."

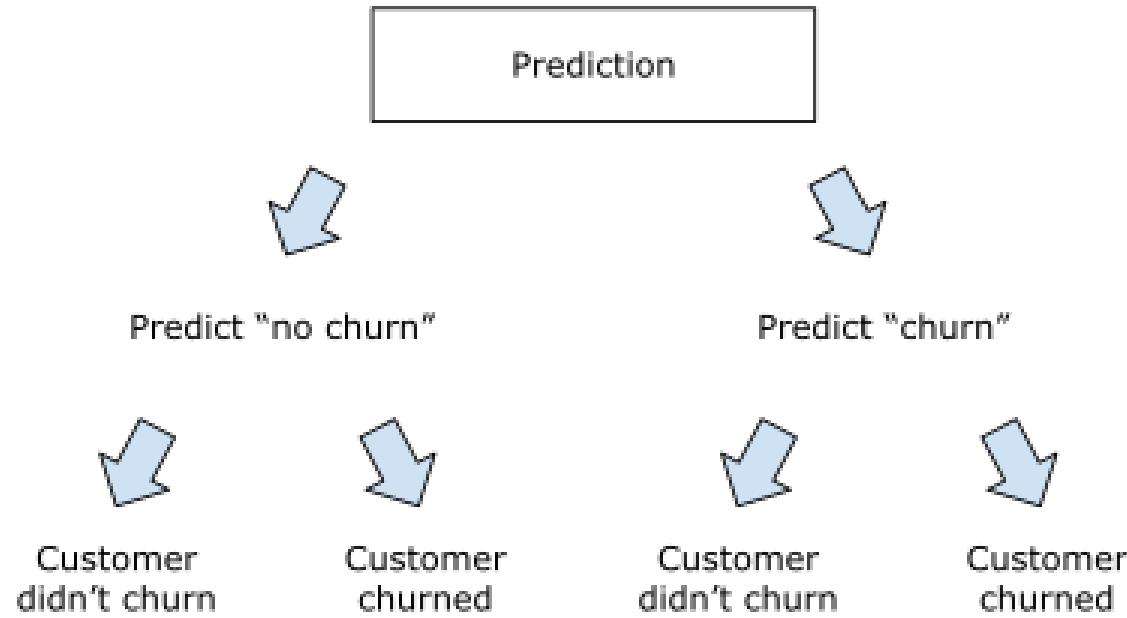
Introduction to confusion table



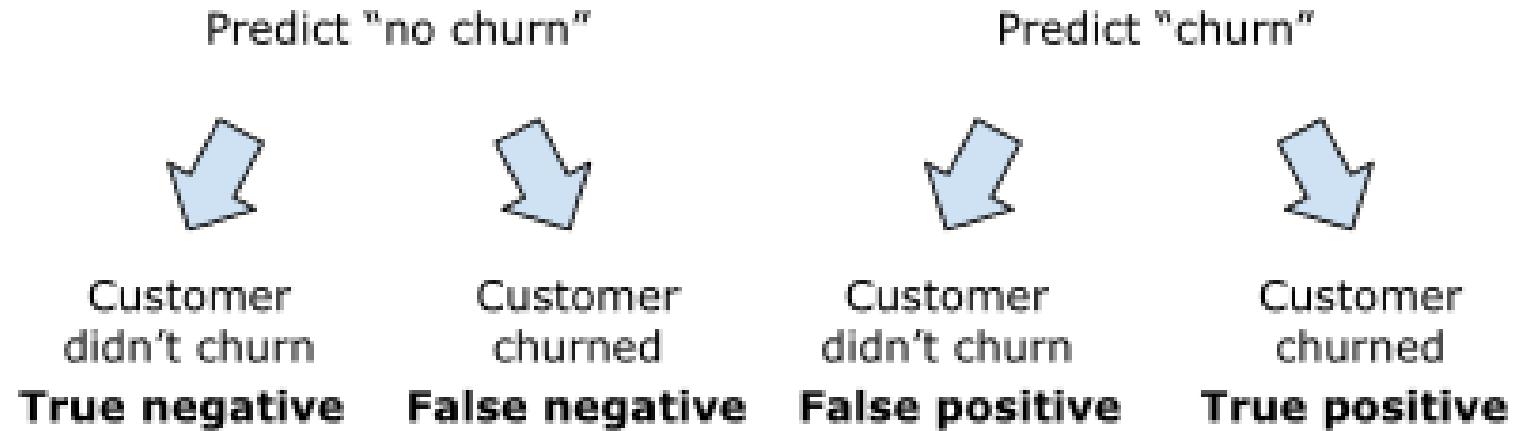
Introduction to confusion table



Introduction to confusion table



Introduction to confusion table



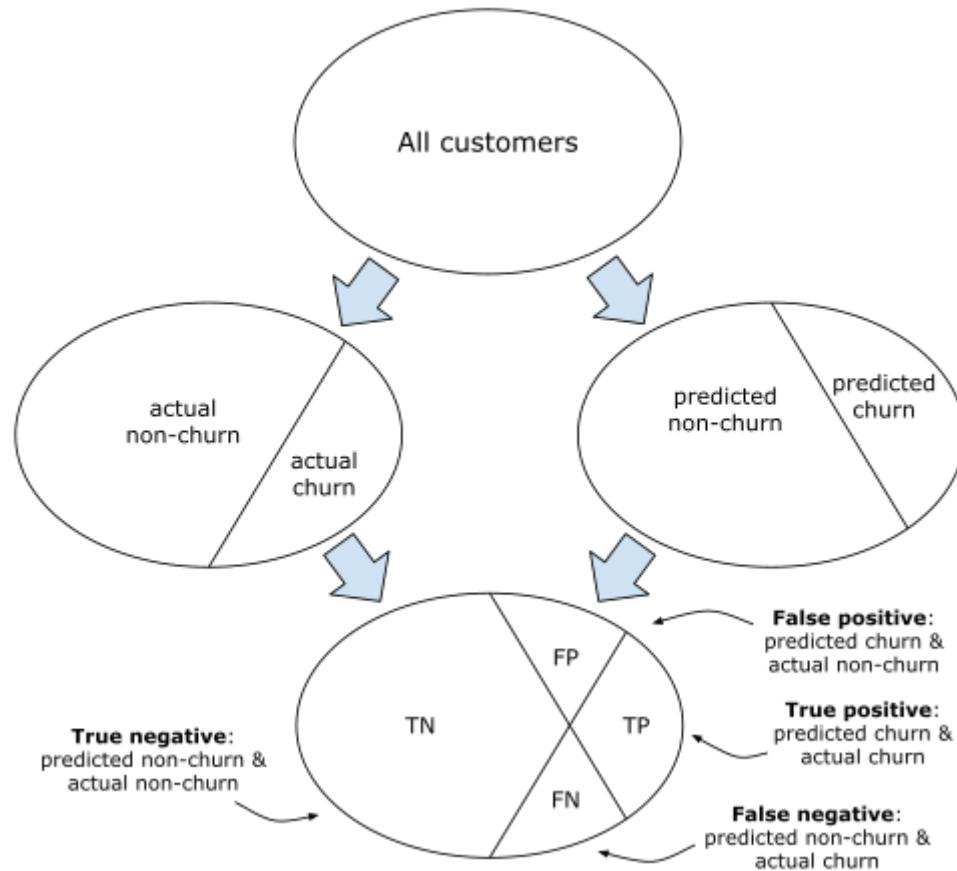
Introduction to confusion table

		Predictions	
		False ("no churn")	True ("churn")
Actual	False ("no churn")	TN	FP
	True ("churn")	FN	TP

Introduction to confusion table

		Predictions	
		False ("no churn")	True ("churn")
Actual	False ("no churn")	1202	172
	True ("churn")	197	289

Calculating the confusion table with NumPy



Calculating the confusion table with NumPy

- Translating these steps to NumPy is straightforward:

```
t = 0.5
predict_churn = (y_pred >= t)
predict_no_churn = (y_pred < t)

actual_churn = (y_val == 1)
actual_no_churn = (y_val == 0)

true_positive = (predict_churn & actual_churn).sum()
false_positive = (predict_churn & actual_no_churn).sum()

false_negative = (predict_no_churn & actual_churn).sum()
true_negative = (predict_no_churn & actual_no_churn).sum()
```

Calculating the confusion table with NumPy

					y_pred					
					0.37 0.51 0.78 0.49 0.29					
y_pred < 0.5						y_pred >= 0.5				
True	False	False	True	True		False	True	True	False	False
predict_no_churn						predict_churn				

Calculating the confusion table with NumPy

					y_val									
					0	1	1	1	0					
					True	False	False	False	True					
					actual_no_churn									
					actual_churn									

Calculating the confusion table with NumPy

- To calculate the number of true positive outcomes in C, we use the logical “and” operator of NumPy (`&`) and the sum method:

```
true_positive = (predict_churn & actual_churn).sum()
```

Calculating the confusion table with NumPy

False	True	True	False	False	predict_churn
-------	------	------	-------	-------	---------------

&

False	True	True	True	False	actual_churn
-------	------	------	------	-------	--------------



False	True	True	False	False
-------	------	------	-------	-------

Calculating the confusion table with NumPy



- As a result, we have the number of true positive cases.
- The other values are computed similarly in lines D, E, and F.

Calculating the confusion table with NumPy

- Now we just need to put all these values together in a NumPy array:

```
confusion_table = np.array(  
    [[true_negative, false_positive],  
     [false_negative, true_positive]])
```

- When we print it, we get the following numbers:

```
[[1202, 172],  
 [ 197, 289]]
```

Calculating the confusion table with NumPy

- The absolute numbers sometimes may be difficult to understand, so we can turn them into fractions by dividing each value by the total number of items:

`confusion_table / confusion_table.sum()`

- This prints the following numbers:

`[[0.646, 0.092],
 [0.105, 0.155]]`

Full model with all features

		Predicted	
		False	True
Actual	False	1202 (65%)	172 (9%)
	True	197 (11%)	289 (15%)

Small model with three features

		Predicted	
		False	True
Actual	False	1189 (63%)	185 (10%)
	True	248 (12%)	238 (13%)



**"Complete
Exercise "**

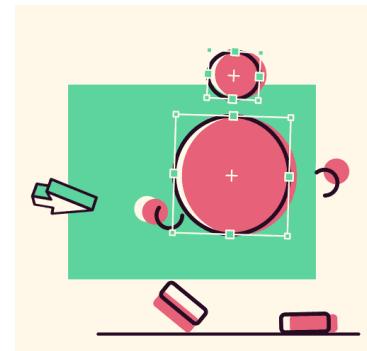
Precision and recall

- In our case it's the number of customers who actually churned (TP), out of all the customers we thought would churn (TP + FP):

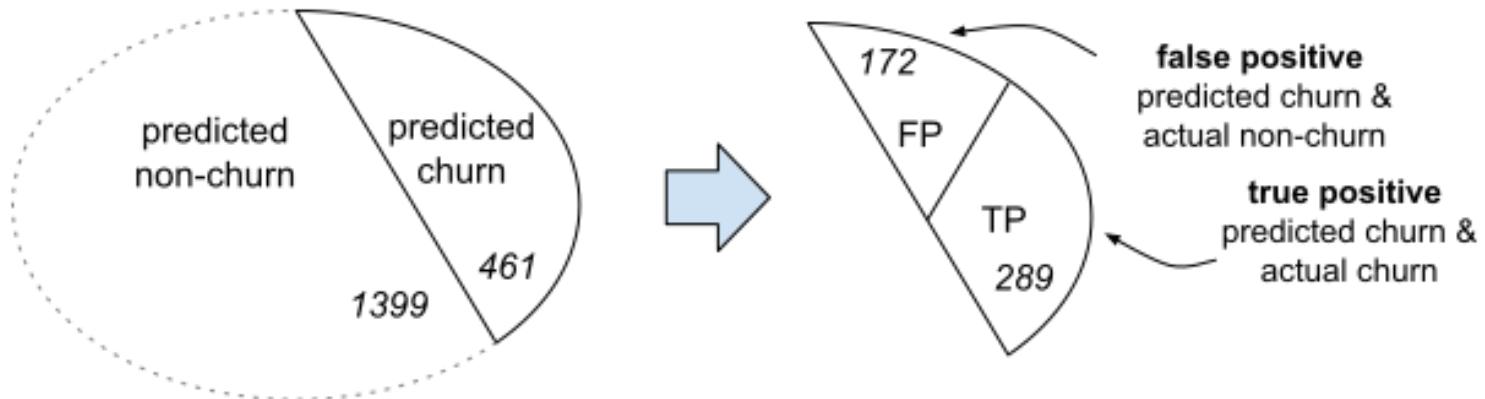
$$P = TP / (TP + FP)$$

- For our model the precision is 62%:

$$P = 289 / (289 + 172) = 172 / 461 = 0.62$$



Precision and recall



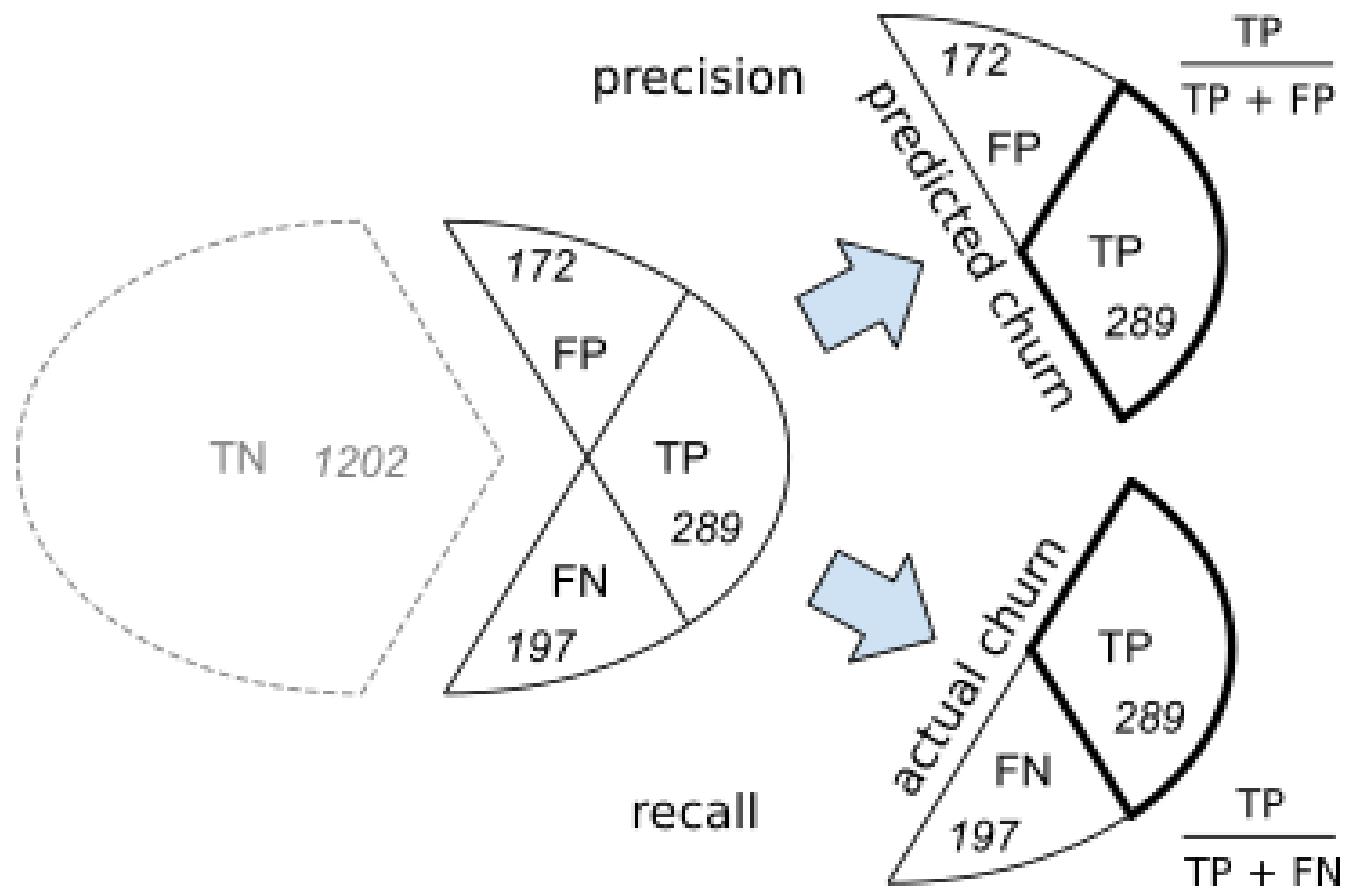
$$\text{precision} = \frac{\text{correct predictions}}{\text{predicted churn}} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{289}{461} = 62\%$$

Precision and recall

- Now we can check the accuracy of this baseline prediction using the same code as previously:

```
accuracy_score(baseline, y_val)
```







**"Complete
Exercises "**

ROC curve and AUC score

- ROC stands for “receiver operating characteristic,” and it was initially designed for evaluating the strength of radar detectors during World War II.
- It was used to assess how well a detector could separate two signals: whether an airplane was there or not.

True positive rate and false positive rate

The ROC curve is based on two quantities, FPR and TPR:

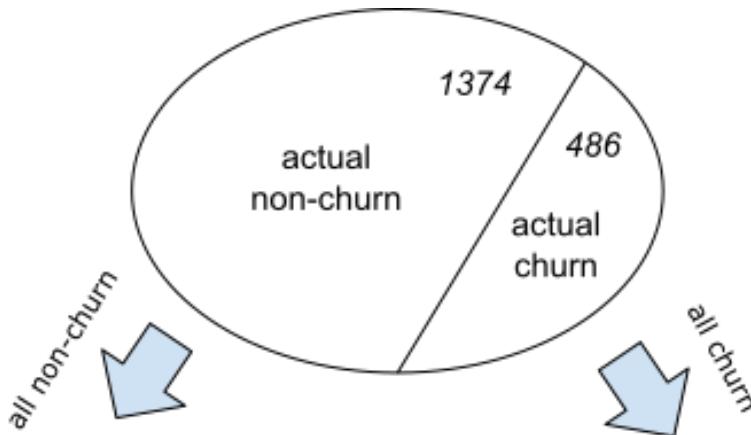
- **False positive rate (FPR)** — The fraction of false positives among all negative examples
- **True positive rate (TPR)** — The fraction of true positives among all positive examples

True positive rate and false positive rate

		Predictions	
		False ("no churn")	True ("churn")
Actual	False ("no churn")	TN	FP
	True ("churn")	FN	TP

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$



FPR

$$\frac{\text{false positive}}{\text{actual non-churn}}$$

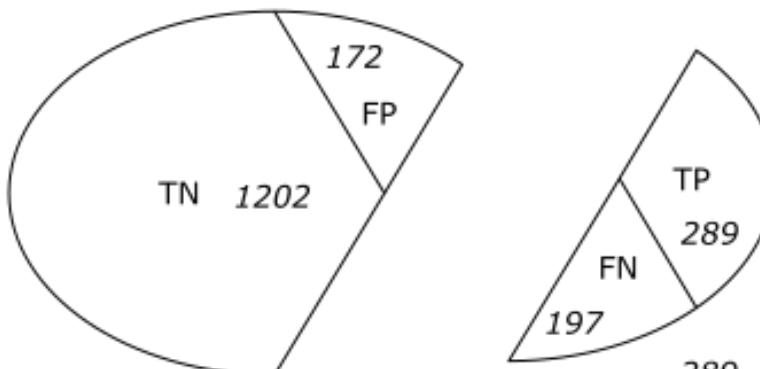
$$\frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\frac{172}{1374} = 12.5\%$$

TPR

$$\frac{\text{true positive}}{\text{actual churn}}$$

$$\frac{\text{TP}}{\text{TP} + \text{FN}}$$



$$\frac{289}{486} = 59\%$$

Evaluating a model at multiple thresholds

- For that, we first iterate over different threshold values and compute the values of the confusion table for each.

```
scores = []
```

```
thresholds = np.linspace(0, 1, 101)
```

```
for t in thresholds:
```

```
    tp = ((y_pred >= t) & (y_val == 1)).sum()
```

```
    fp = ((y_pred >= t) & (y_val == 0)).sum()
```

```
    fn = ((y_pred < t) & (y_val == 1)).sum()
```

```
    tn = ((y_pred < t) & (y_val == 0)).sum()
```

```
    scores.append((t, tp, fp, fn, tn))
```

Evaluating a model at multiple thresholds

- It's not easy to deal with a list of tuples, so let's convert it to a Pandas dataframe:

```
df_scores = pd.DataFrame(scores)
df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn']
```

```
df_scores[::10]
```

threshold	tp	fp	fn	tn
0	0.0	486	1374	0
10	0.1	458	726	28
20	0.2	421	512	65
30	0.3	380	350	106
40	0.4	337	257	149
50	0.5	289	172	197
60	0.6	200	105	286
70	0.7	99	34	387
80	0.8	7	1	479
90	0.9	0	0	486
100	1.0	0	0	486

Evaluating a model at multiple thresholds

- Now we can compute the TPR and FPR scores.
- Because the data is now in a dataframe, we can do it for all the values at once:

```
df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)  
df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)
```

```
df_scores[::10]
```

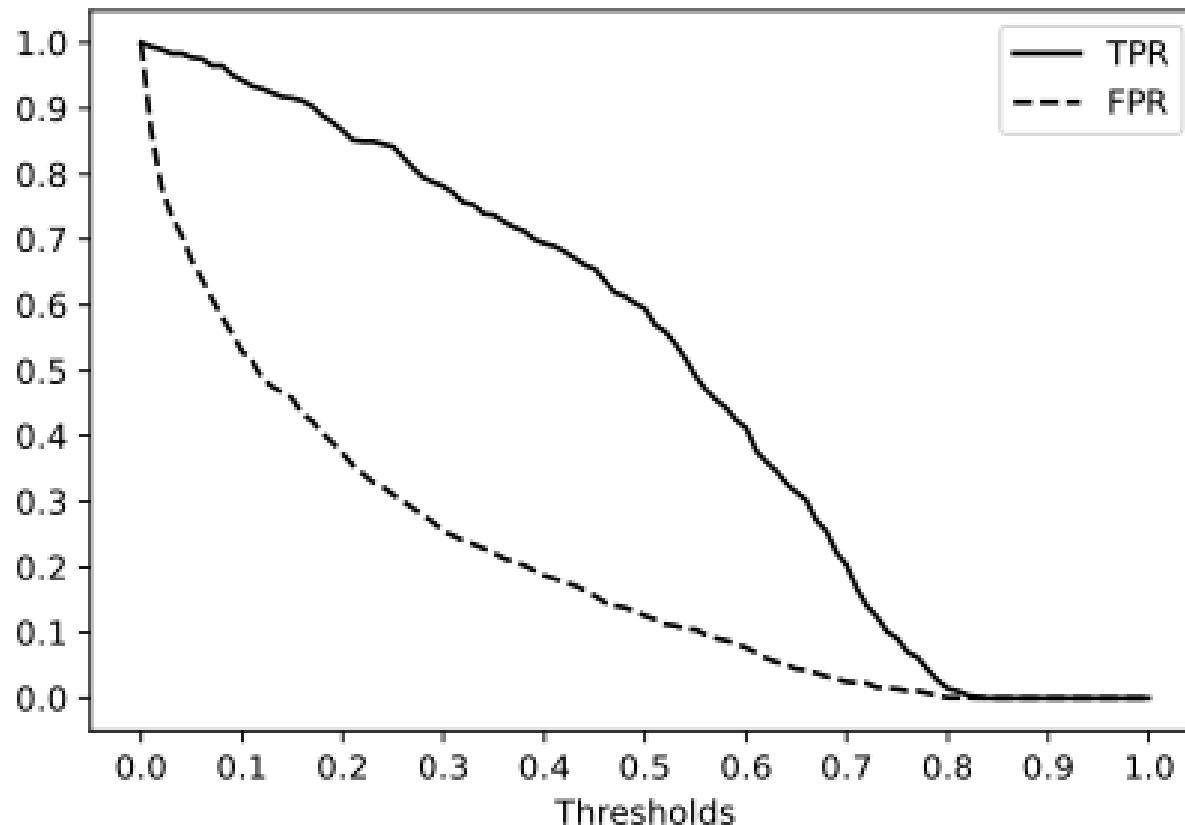
threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000
10	0.1	458	726	28	648	0.942387
20	0.2	421	512	65	862	0.866255
30	0.3	380	350	106	1024	0.781893
40	0.4	337	257	149	1117	0.693416
50	0.5	289	172	197	1202	0.594650
60	0.6	200	105	286	1269	0.411523
70	0.7	99	34	387	1340	0.203704
80	0.8	7	1	479	1373	0.014403
90	0.9	0	0	486	1374	0.000000
100	1.0	0	0	486	1374	0.000000

Evaluating a model at multiple thresholds

- Let's plot them (next figure):

```
plt.plot(df_scores.threshold, df_scores.tpr, label='TPR')
plt.plot(df_scores.threshold, df_scores.fpr, label='FPR')
plt.legend()
```

TPR and FPR



Random baseline model

- A random model outputs a random score between 0 and 1 regardless of the input.
- It's easy to implement: we simply generate an array with uniform random numbers:

```
np.random.seed(1)  
y_rand = np.random.uniform(0, 1, size=len(y_val))
```

```
def tpr_fpr_dataframe(y_val, y_pred):
    scores = []

    thresholds = np.linspace(0, 1, 101)

    for t in thresholds:
        tp = ((y_pred >= t) & (y_val == 1)).sum()
        fp = ((y_pred >= t) & (y_val == 0)).sum()
        fn = ((y_pred < t) & (y_val == 1)).sum()
        tn = ((y_pred < t) & (y_val == 0)).sum()
        scores.append((t, tp, fp, fn, tn))

    df_scores = pd.DataFrame(scores)
    df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn']

    df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
    df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)

    return df_scores
```

Random baseline model

- Now let's use this function to calculate the TPR and FPR for the random model:

```
df_rand = tpr_fpr_dataframe(y_val,  
y_rand)
```

- This creates a dataframe with TPR and FPR values at different thresholds

```
np.random.seed(1)  
y_rand = np.random.uniform(0, 1, size=len(y_val))  
df_rand = tpr_fpr_dataframe(y_val, y_rand)  
df_rand[::10]
```

threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000
10	0.1	440	1236	46	138	0.905350
20	0.2	392	1101	94	273	0.806584
30	0.3	339	972	147	402	0.697531
40	0.4	288	849	198	525	0.592593
50	0.5	239	723	247	651	0.491770
60	0.6	193	579	293	795	0.397119
70	0.7	152	422	334	952	0.312757
80	0.8	98	302	388	1072	0.201646
90	0.9	57	147	429	1227	0.117284
100	1.0	0	0	486	1374	0.000000

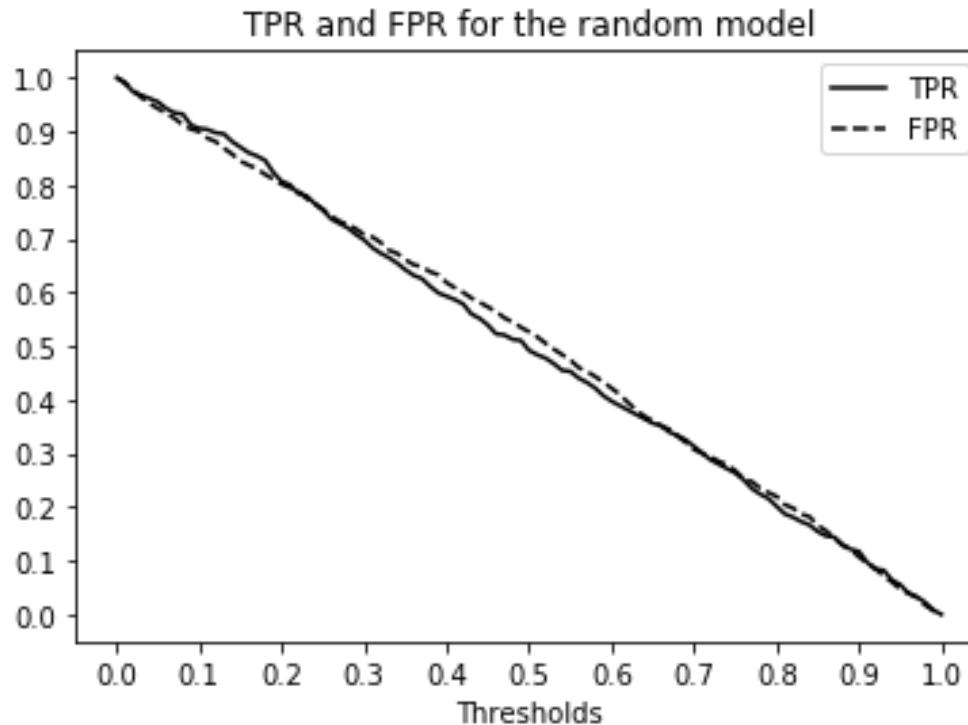
Random baseline m

- Let's plot them:

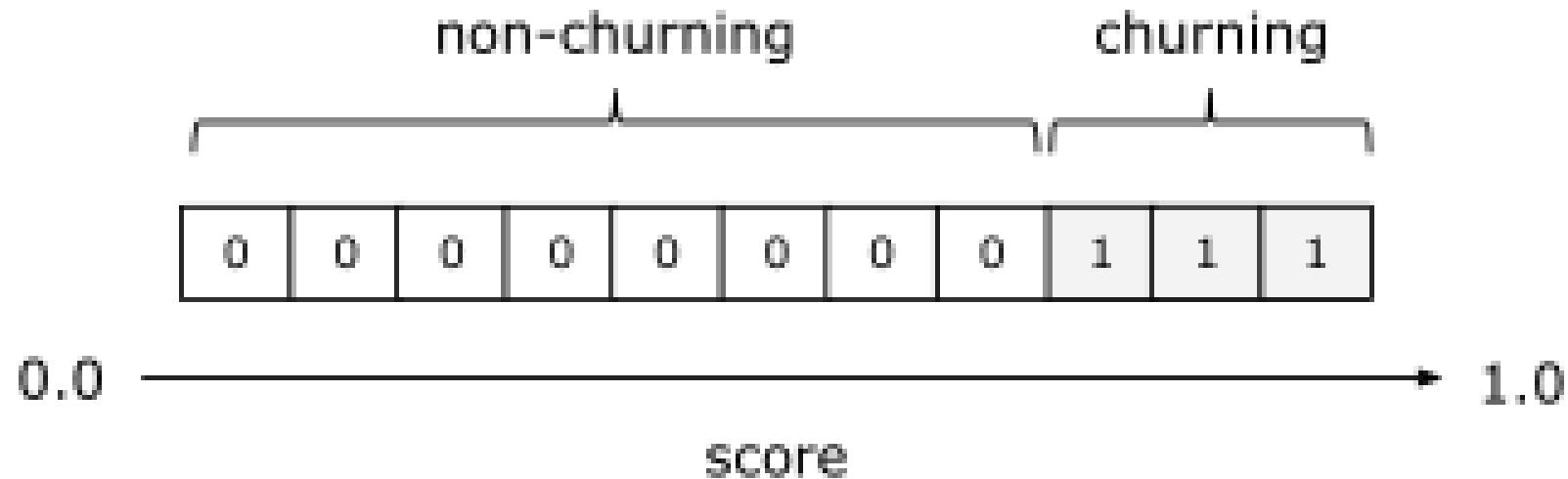
```
plt.plot(df_rand.threshold, df_rand.tpr, label='TPR')
plt.plot(df_rand.threshold, df_rand.fpr, label='FPR')
plt.legend()
```



Random baseline model



The ideal model



The ideal model

- Let's do it:

```
num_neg = (y_val == 0).sum()  
num_pos = (y_val == 1).sum()
```

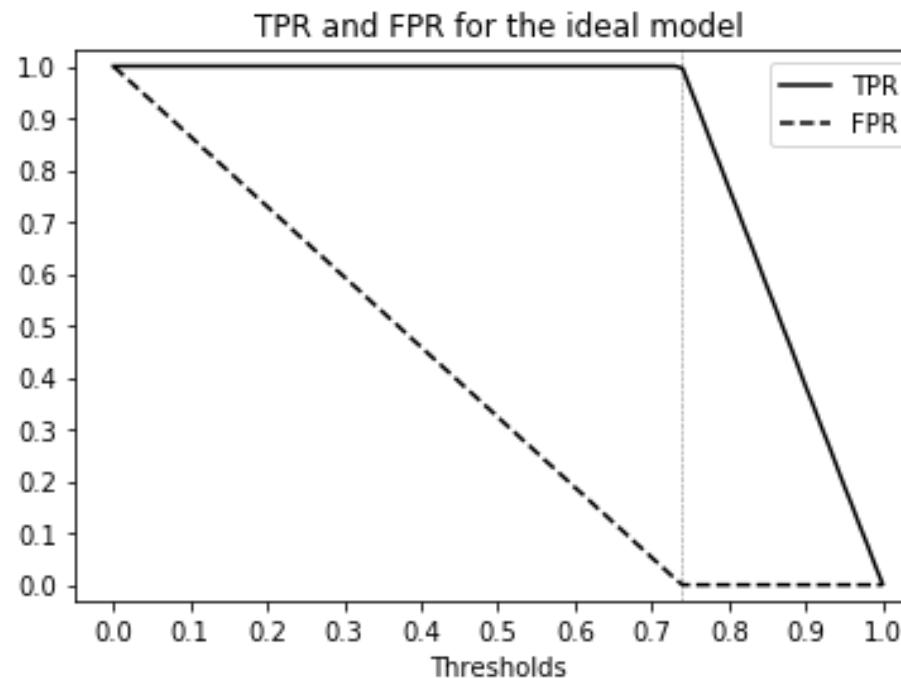
```
y_ideal = np.repeat([0, 1], [num_neg, num_pos])  
y_pred_ideal = np.linspace(0, 1, num_neg + num_pos)
```

```
df_ideal = tpr_fpr_dataframe(y_ideal, y_pred_ideal)
```

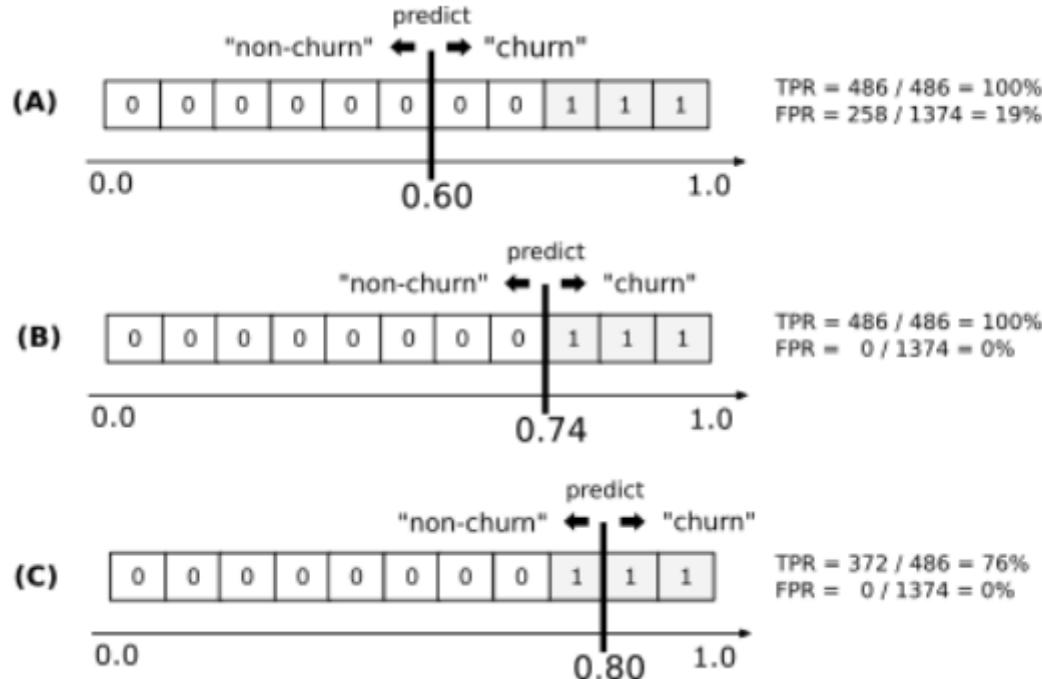
threshold		tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	486	1188	0	186	1.000000	0.864629
20	0.2	486	1002	0	372	1.000000	0.729258
30	0.3	486	816	0	558	1.000000	0.593886
40	0.4	486	630	0	744	1.000000	0.458515
50	0.5	486	444	0	930	1.000000	0.323144
60	0.6	486	258	0	1116	1.000000	0.187773
70	0.7	486	72	0	1302	1.000000	0.052402
80	0.8	372	0	114	1374	0.765432	0.000000
90	0.9	186	0	300	1374	0.382716	0.000000
100	1.0	1	0	485	1374	0.002058	0.000000

- Now we can plot it (figure):

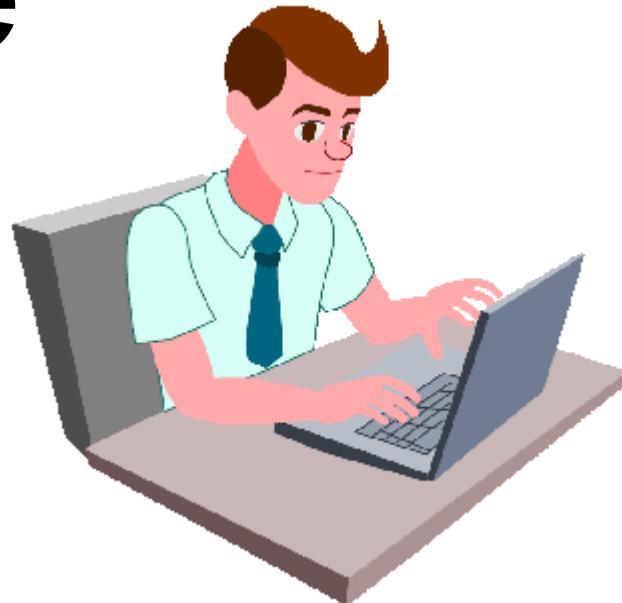
```
plt.plot(df_ideal.threshold, df_ideal.tpr, label='TPR')
plt.plot(df_ideal.threshold, df_ideal.fpr, label='FPR')
plt.legend()
```



The ideal model



"Complete Exercise "



ROC Curve

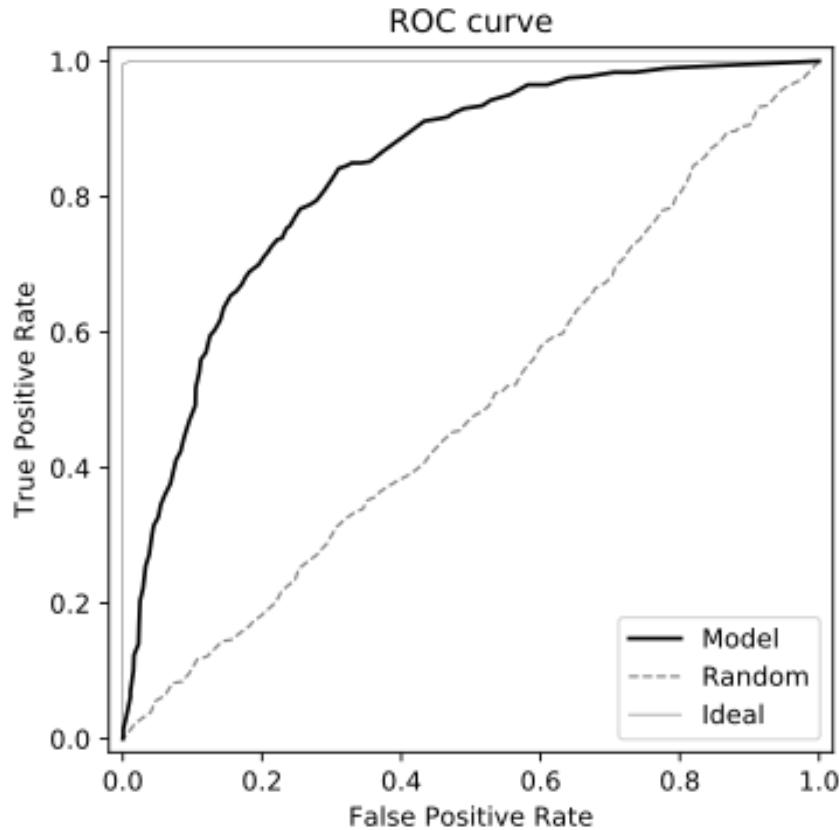
```
plt.figure(figsize=(5, 5))
```

```
plt.plot(df_scores.fpr, df_scores.tpr, label='Model')  
plt.plot(df_rand.fpr, df_rand.tpr, label='Random')  
plt.plot(df_ideal.fpr, df_ideal.tpr, label='Ideal')
```

```
plt.legend()
```

ROC Curve

- As a result, we get a ROC curve



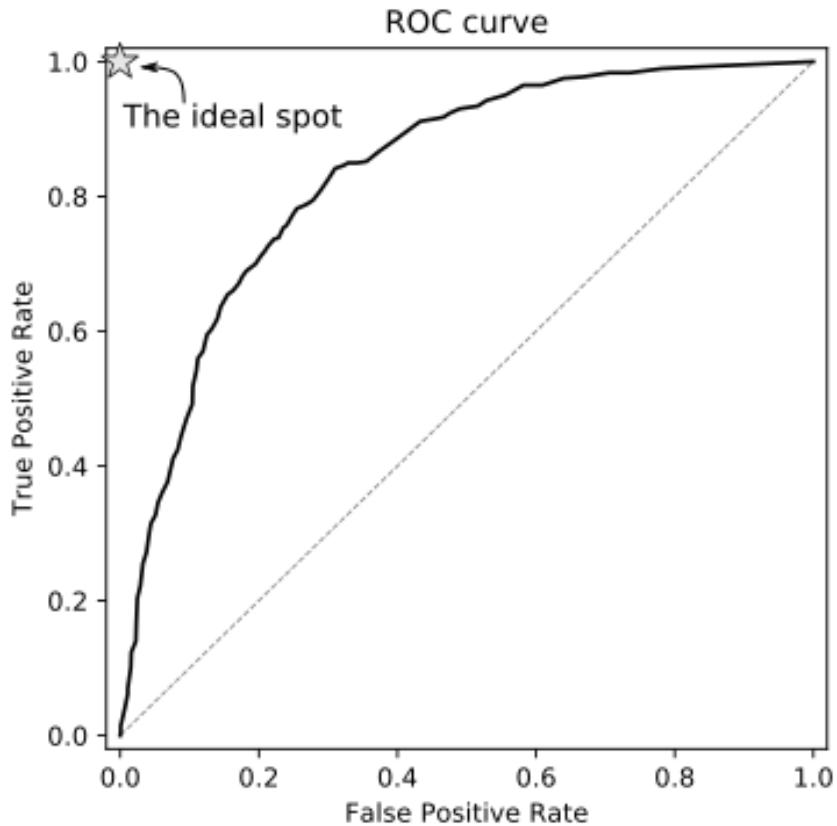
ROC Curve

- With this information, we can reduce the code for plotting the curve to the following:

```
plt.figure(figsize=(5, 5))
plt.plot(df_scores.fpr, df_scores.tpr)
plt.plot([0, 1], [0, 1])
```

ROC Curve

- Produces the result in figure



ROC Curve

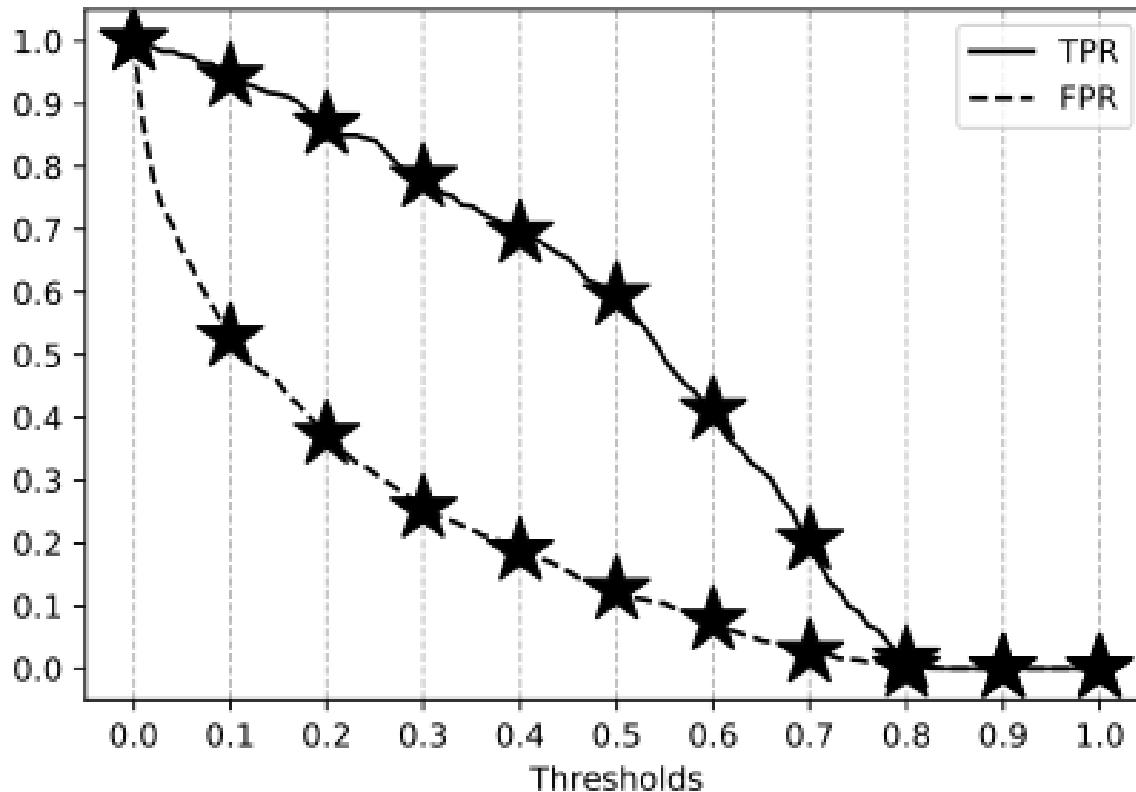
- We simply can use the `roc_curve` function from the `metrics` package of Scikit-Learn:

```
from sklearn.metrics import roc_curve
```

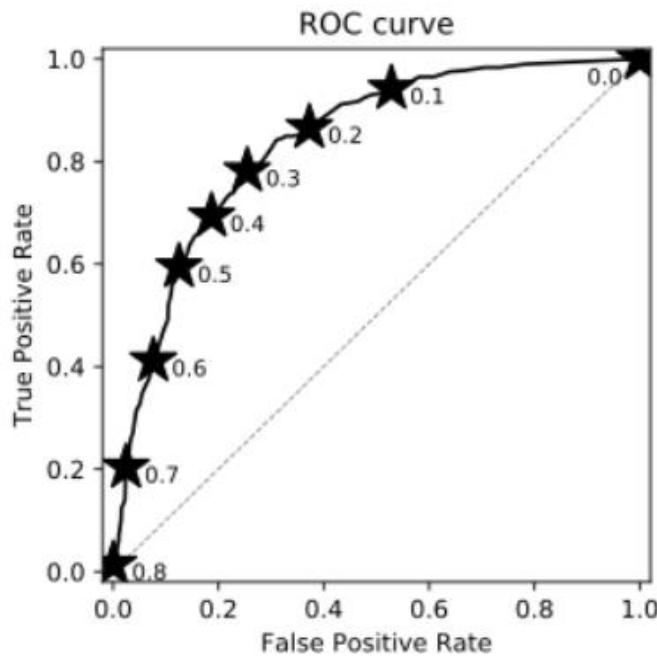
```
fpr, tpr, thresholds = roc_curve(y_val, y_pred)
```

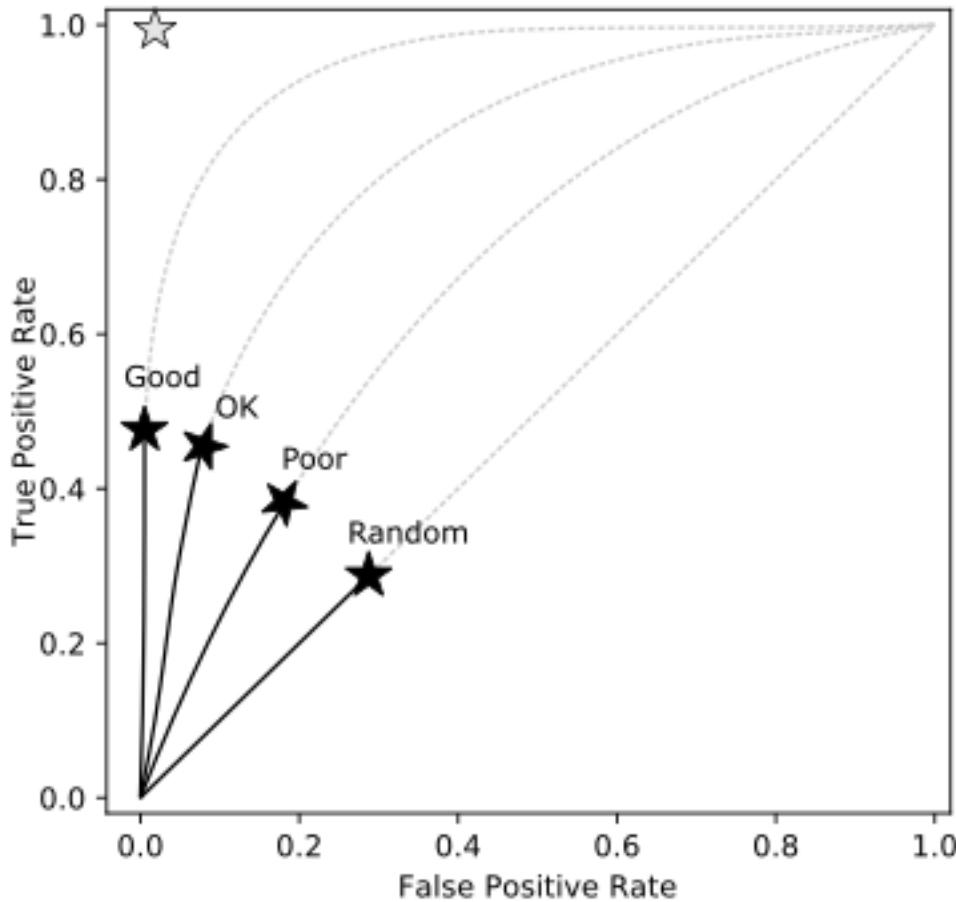
```
plt.figure(figsize=(5, 5))  
plt.plot(fpr, tpr)  
plt.plot([0, 1], [0, 1])
```

TPR and FPR



threshold	fpr	tpr
100	1.0	0.000000
90	0.9	0.000000
80	0.8	0.000728
70	0.7	0.024745
60	0.6	0.076419
50	0.5	0.125182
40	0.4	0.187045
30	0.3	0.254731
20	0.2	0.372635
10	0.1	0.528384
0	0.0	1.000000



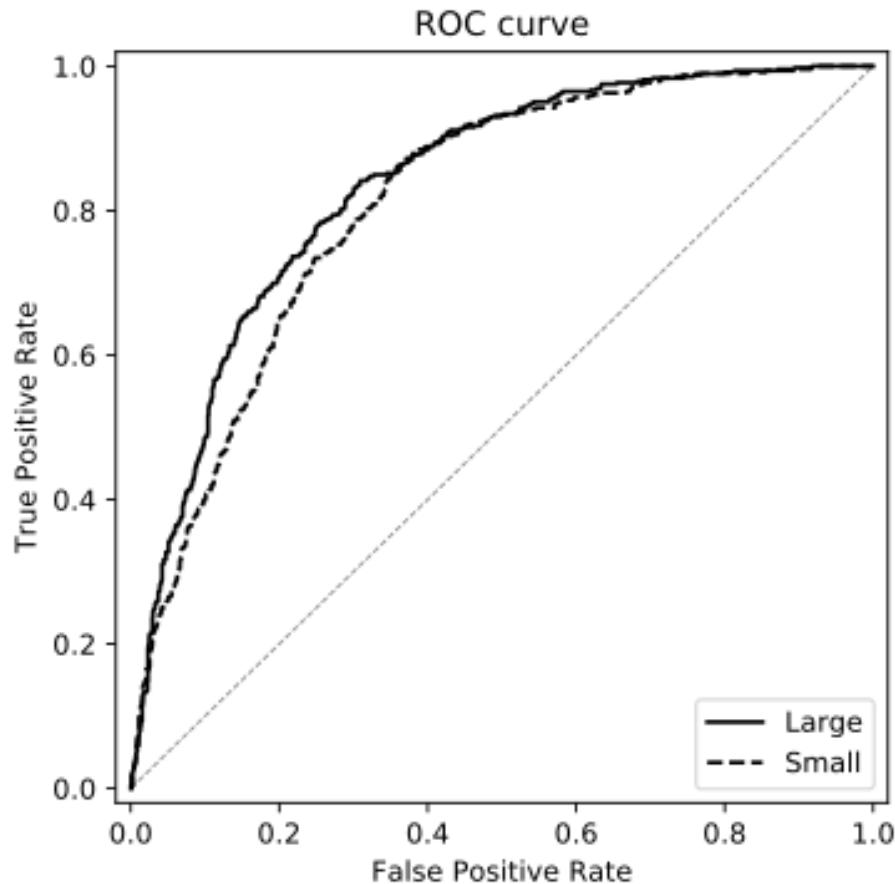


ROC Curve

```
fpr_large, tpr_large, _ = roc_curve(y_val, y_pred)  
fpr_small, tpr_small, _ = roc_curve(y_val, y_pred_small)
```

```
plt.figure(figsize=(5, 5))
```

```
plt.plot(fpr_large, tpr_large, color='black', label='Large')  
plt.plot(fpr_small, tpr_small, color='black', label='Small')  
plt.plot([0, 1], [0, 1])  
plt.legend()
```



Area under the ROC curve (AUC)

- To calculate the AUC for our models we can use `auc`, a function from the `metrics` package of Scikit-Learn:

```
from sklearn.metrics import auc  
auc(df_scores.fpr, df_scores.tpr)
```

Area under the ROC curve (AUC)

- For the large model, the result is 0.84; for the small model, it's 0.81 (figure).
- Churn prediction is a complex problem, so an AUC of 80% is quite good.

```
from sklearn.metrics import auc  
auc(df_scores.fpr, df_scores.tpr)
```

```
0.8359001084215382
```

```
auc(df_scores_small.fpr, df_scores_small.tpr)
```

```
0.8125475467380692
```

Area under the ROC curve (AUC)

- If all we need is the AUC, we don't need to compute the ROC curve first.
- We can take a shortcut and use a function from Scikit-Learn that takes care of everything and simply returns the AUC of our model:

```
from sklearn.metrics import roc_auc_score  
roc_auc_score(y_val, y_pred)
```

Area under the ROC curve (AUC)

- We get approximately the same results as previously

```
from sklearn.metrics import roc_auc_score  
roc_auc_score(y_val, y_pred)
```

```
0.8363396349608545
```

```
roc_auc_score(y_val, y_pred_small)
```

```
0.8129354083179088
```

Area under the ROC curve (AUC)

```
neg = y_pred[y_val == 0]  
pos = y_pred[y_val == 1]
```

```
np.random.seed(1)  
neg_choice = np.random.randint(low=0, high=len(neg),  
size=10000)  
pos_choice = np.random.randint(low=0, high=len(pos),  
size=10000)  
(pos[pos_choice] > neg[neg_choice]).mean()
```

Parameter tuning

- It tells us how well the model will perform on these specific data points.
- However, it doesn't necessarily mean it will perform equally well on other data points. So how do we check if the model indeed works well in a consistent and predictable manner?

K-fold cross-validation



K-fold cross-validation

```
def train(df, y):
    cat = df[categorical + numerical].to_dict(orient='rows')

    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear')
    model.fit(X, y)

    return dv, model
```

K-fold cross-validation

- We apply the vectorizer to the dataframe, get a matrix and finally apply the model to the matrix to get predictions:

```
def predict(df, dv, model):
    cat = df[categorical + numerical].to_dict(orient='rows')

    X = dv.transform(cat)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred
```

```
from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=1)

aucs = []

for train_idx, val_idx in kfold.split(df_train_full):
    df_train = df_train_full.iloc[train_idx]
    df_val = df_train_full.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values

    dv, model = train(df_train, y_train)
    y_pred = predict(df_val, dv, model)

    auc = roc_auc_score(y_val, y_pred)
    aucs.append(auc)
```

K-fold cross-validation

- We used K-fold cross-validation with K=10.
- Thus, when we run it, at the end we get 10 different numbers — 10 AUC scores evaluated on 10 different validation folds:

0.849, 0.841, 0.859, 0.833, 0.824, 0.841, 0.844, 0.822, 0.845,
0.861

K-fold cross-validation

- It's not a single number anymore, and we can think of it as a distribution of AUC scores for our model.
- So we can get some statistics from this distribution, such as the mean and standard deviation:

```
print('auc = %0.3f ± %0.3f' % (np.mean(aucs), np.std(aucs)))
```

Finding best parameters

- We first adjust the train function to take in an additional parameter:

```
def train(df, y, C):
    cat = df[categorical + numerical].to_dict(orient='rows')

    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear', C=C)
    model.fit(X, y)

    return dv, model
```

```
nfolds = 5
kfold = KFold(n_splits=nfolds, shuffle=True, random_state=1)

for C in [0.001, 0.01, 0.1, 0.5, 1, 10]:
    aucss = []

    for train_idx, val_idx in kfold.split(df_train_full):
        df_train = df_train_full.iloc[train_idx]
        df_val = df_train_full.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        aucss.append(auc)

    print('C=%s, auc = %0.3f ± %0.3f' % (C, np.mean(aucss), np.std(aucss)))
```

Finding best parameters

- When we run it, it prints:

C=0.001, auc = 0.825 ± 0.013

C=0.01, auc = 0.839 ± 0.009

C=0.1, auc = 0.841 ± 0.008

C=0.5, auc = 0.841 ± 0.007

C=1, auc = 0.841 ± 0.007

C=10, auc = 0.841 ± 0.007

Finding best parameters

- Let's use our train and predict functions for that:

```
y_train = df_train_full.churn.values
```

```
y_test = df_test.churn.values
```

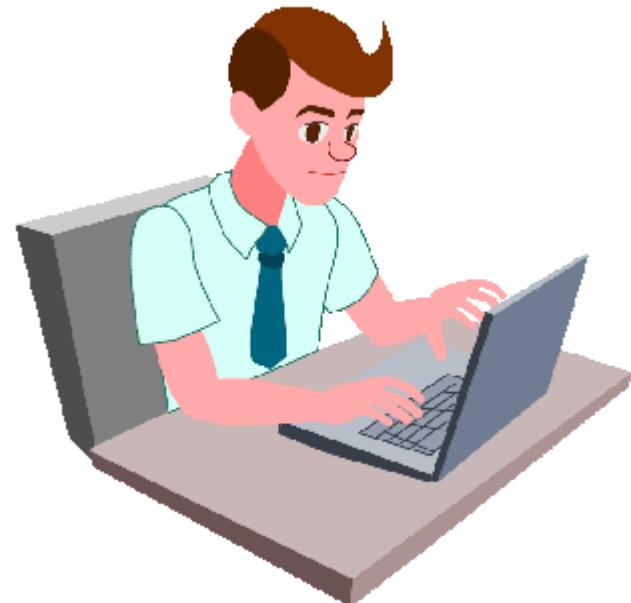
```
dv, model = train(df_train_full, y_train, C=0.5)
```

```
y_pred = predict(df_test, dv, model)
```

```
auc = roc_auc_score(y_test, y_pred)
```

```
print('auc = %.3f' % auc)
```

"Complete Exercises & Lab"



Summary

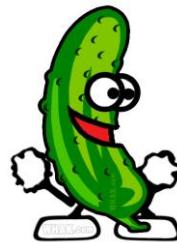


- A metric is a single number that can be used for evaluating the performance of a machine learning model.
- Once we choose a metric, we can use it to compare multiple machine learning models with each other and select the best one.
- Accuracy is the simplest binary classification metric: it tells us the percentage of correctly classified observations in the validation set.

3. Creating a Model Service

What you will learn

- Saving models with Pickle
- Serving models with Flask
- Managing dependencies with Pipenv
- Making the service self-contained with Docker
- Deploying it to the cloud using AWS Elastic Beanstalk



Using the Model

Churn Prediction Model

- We will use the previously created CUSTOMER CHURN Model with the Jupyter Notebook



Will this customer leave?

```
customer = {  
    'customerid': '8879-zkjof',  
    'gender': 'female',  
    'seniorcitizen': 0,  
    'partner': 'no',  
    'dependents': 'no',  
    'tenure': 41,  
    'phoneservice': 'yes',  
    'multiplelines': 'no',  
    'internetservice': 'dsl',  
    'onlinesecurity': 'yes',  
    'onlinebackup': 'no',  
    'deviceprotection': 'yes',  
    'techsupport': 'yes',  
    'streamingtv': 'yes',  
    'streamingmovies': 'yes',  
    'contract': 'one_year',  
    'paperlessbilling': 'yes',  
    'paymentmethod':  
        'bank_transfer_(automatic)',  
    'monthlycharges': 79.85,  
    'totalcharges': 3320.75,  
}
```

-

Will this customer leave?

- To predict whether this customer is going to churn, we can use the predict function we wrote in the previous lesson:

```
df = pd.DataFrame([customer])
y_pred = predict(df, dv, model)
y_pred[0]
```

OUTPUT: 0.059605



Prediction with an entire Pandas DataFrame

- Now let's take a look at the predict function, which we wrote previously for applying the model to the customers in the validation set:

```
def predict(df, dv, model):  
    cat = df[categorical + numerical].to_dict(orient='rows')  
    X = dv.transform(cat)  
    y_pred = model.predict_proba(X)[:, 1]  
    return y_pred
```

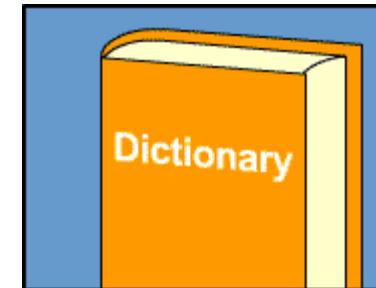


Predictions with a Dictionary

- To avoid doing this unnecessary conversion, we can create a separate function for predicting the probability of churn for a single customer only. Let's call this function predict_single:

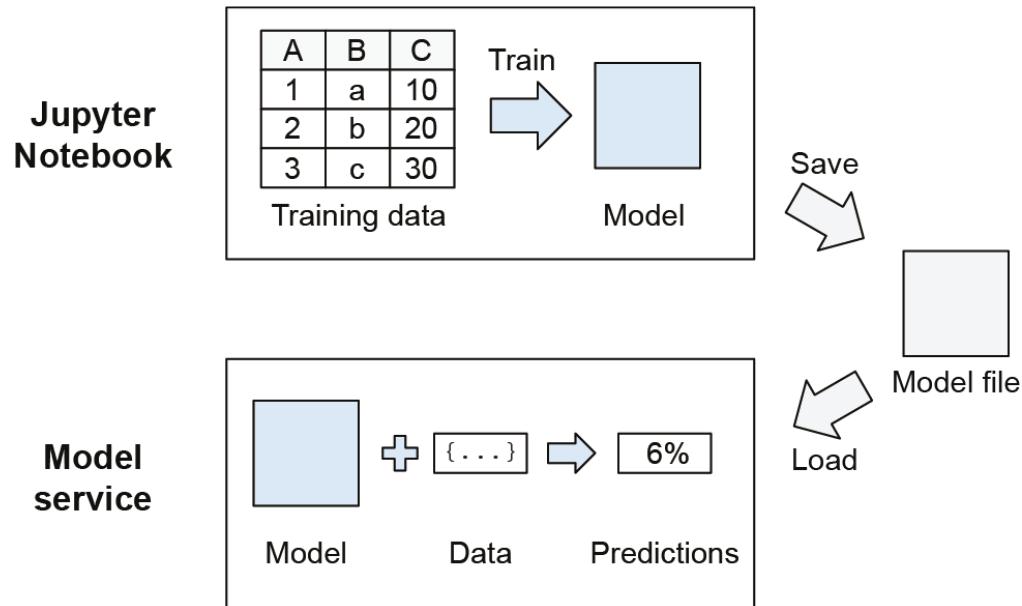
```
def predict_single(customer, dv, model):  
    X = dv.transform([customer])  
    y_pred = model.predict_proba(X)[:, 1]  
    return y_pred[0]
```

```
predict_single(customer, dv, model)
```



Using Pickle to Save and Load the Model

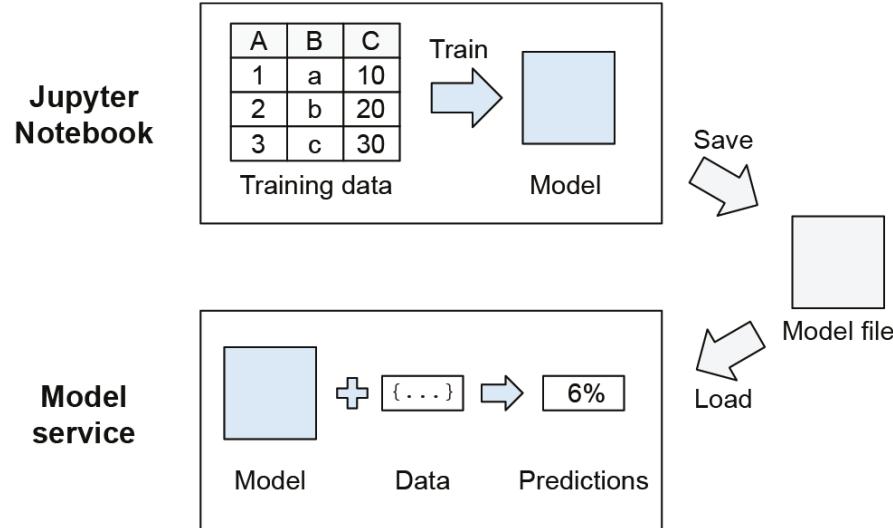
To be able to use it outside of our notebook, we need to save it, and then later, another process can load and use it



Save the Model

```
import pickle
```

```
with open('churn-model.bin', 'wb') as f_out:  
    pickle.dump((dv, model), f_out)
```

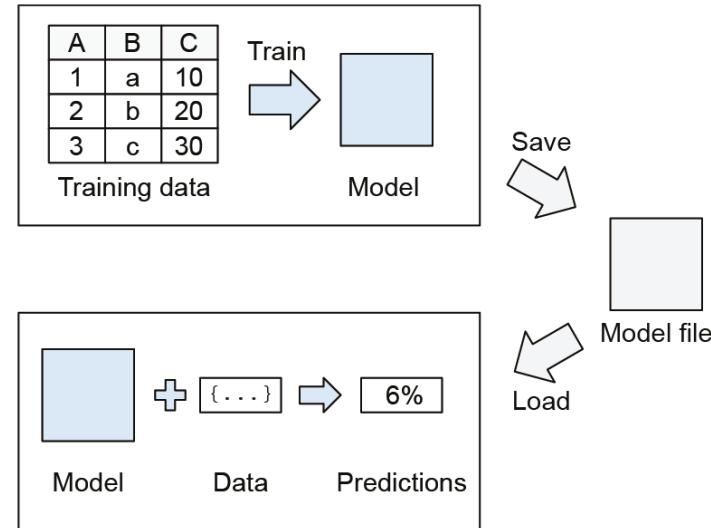


Load the Model

```
import pickle
```

```
with open('churn-model.bin', 'rb') as f_in:  
    dv, model = pickle.load(f_in)
```

Jupyter
Notebook



Load the Model and Apply the Customer

Let's create a simple Python script that loads the model and applies it to a customer.

- The predict_single function that we wrote earlier
- The code for loading the model
- The code for applying the model to a customer

Imports

```
import pickle  
import numpy as np
```

Create the predict_single function

```
def predict_single(customer, dv, model):  
    X = dv.transform([customer])  
    y_pred = model.predict_proba(X)[:, 1]  
    return y_pred[0]
```

Load the Model

```
with open('churn-model.bin', 'rb') as f_in:  
    dv, model = pickle.load(f_in)
```

Apply the Customer

```
customer = {  
    'customerid': '8879-zkjof',  
    'gender': 'female',  
    'seniorcitizen': 0,  
    'partner': 'no',  
    'dependents': 'no',  
    'tenure': 41,  
    'phoneservice': 'yes',  
    'multiplelines': 'no',  
    'internetservice': 'dsl',  
    'onlinesecurity': 'yes',  
    'onlinebackup': 'no',  
    'deviceprotection': 'yes',  
    'techsupport': 'yes',  
    'streamingtv': 'yes',  
    'streamingmovies': 'yes',  
    'contract': 'one_year',  
    'paperlessbilling': 'yes',  
    'paymentmethod': 'bank_transfer_(automatic)',  
    'monthlycharges': 79.85,  
    'totalcharges': 3320.75,  
}
```

```
prediction = predict_single(customer, dv, model)
```

Display the Results

```
print('prediction: %.3f' % prediction)
```

```
if prediction >= 0.5:  
    print('verdict: Churn')  
else:  
    print('verdict: Not churn')
```

Run the Script and Validate the Results

```
python churn_serving.py
```

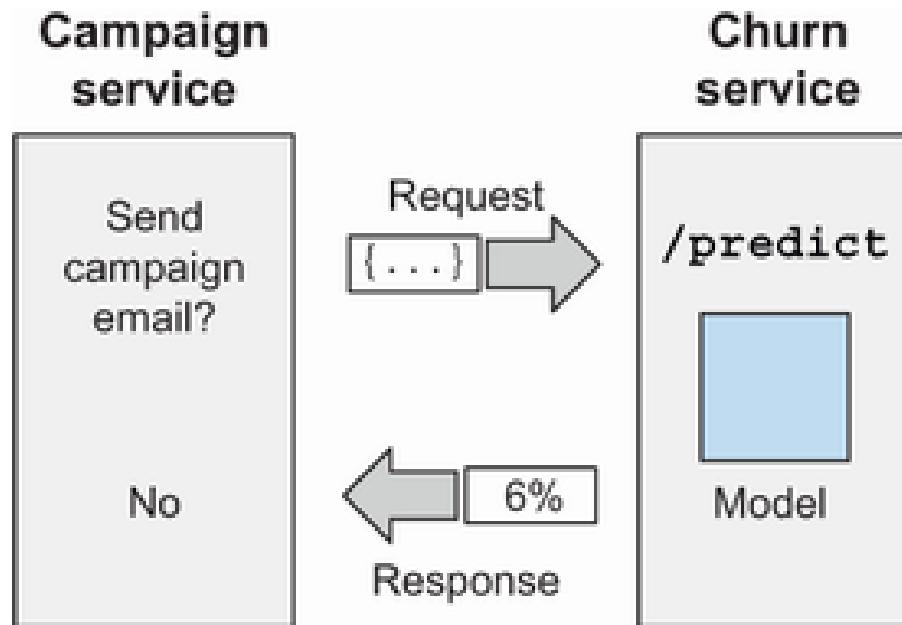
```
prediction: 0.059
```

```
verdict: Not churn
```

Model Serving

Web Service

- Microservices!



Flask

- The easiest way to implement a web service in Python is to use Flask. It's quite lightweight, requires little code to get started, and hides most of the complexity of dealing with HTTP requests and responses.



4. Neural networks and deep learning

Overview

This lesson covers

- Convolutional neural networks for image classification
- TensorFlow and Keras—frameworks for building neural networks
- Using pre trained neural networks
- Internals of a convolutional neural network
- Training a model with transfer learning
- Data augmentations—the process of generating more training data

Fashion classification

- Imagine that we work at an online fashion marketplace.
- Our users upload thousands of images every day to sell their clothes.
- We want to help our users create listings faster by automatically recommending the right category for their clothes.
- To do it, we need a model for classifying images. Previously, we covered multiple models for classification: logistic regression, decision trees, random forests, and gradient boosting.

GPU vs. CPU

- Training a neural network is a computationally demanding process, and it requires powerful hardware to make it faster.
- To speed up training, we usually use GPUs—graphical processing units, or, simply, graphic cards.
- For this lesson, a GPU is not required. You can do everything on your laptop, but without a GPU, it will be approximately eight times slower than with a GPU.

Downloading the clothing dataset

- For this project, we need a dataset of clothes. We will use a subset of the clothing dataset (for more information, check <https://github.com/fenago/clothing-dataset>), which contains around 3,800 images of 10 different classes.
- The data is available in a GitHub repository. Let's clone it:

```
!git clone https://github.com/fenago/clothing-dataset-small-  
master.git
```

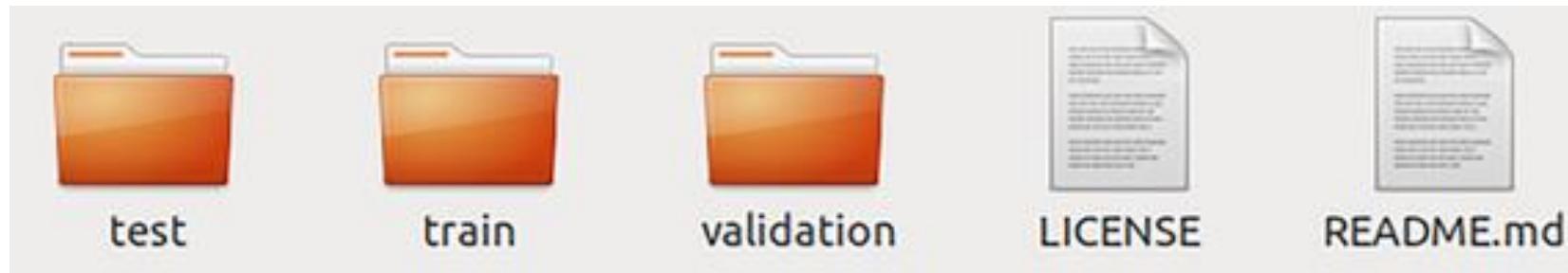
Downloading the clothing dataset

```
!git clone https://github.com/fenago/clothing-dataset-small-master.git
```

Downloading the clothing dataset

The dataset is already split into folders (figure):

- train: Images for training a model (3,068 images)
- validation: Images for validating (341 image)
- test: Images for testing (372 images)



Downloading the clothing dataset

- Each of these folders has 10 subfolders: one subfolder for each type of clothing



Downloading the clothing dataset

- Each subfolder contains images of only one class



TensorFlow and Keras

- Use pip to do it:

```
pip install tensorflow
```

- We're ready to start and create a new notebook called lesson-07-neural-nets. As usual, we begin by importing NumPy and Matplotlib:

```
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

TensorFlow and Keras

- Next, import TensorFlow and Keras:

```
import tensorflow as tf  
from tensorflow import keras
```

Images

- Keras offers a special function for loading images called `load_img`.
Let's import it:

```
from tensorflow.keras.preprocessing.image import load_img
```

- Let's use this function to take a look at one of the images:

```
path = './clothing-dataset-small/train/t-shirt'  
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'  
fullname = path + '/' + name  
load_img(fullname)
```

images

```
path = './clothing-dataset-small/train/t-shirt'  
name = '5f0a3fa0-6a3d-4b68-b213-72766a643de7.jpg'  
fullname = path + '/' + name  
load_img(fullname)
```



images

- To resize the image, specify the target_size parameter:

```
load_img(fullname, target_size=(299,  
299))
```

- As a result, the image becomes square and a bit squashed

```
load_img(fullname, target_size=(299, 299))
```



Convolutional neural networks

- Neural networks are a class of machine learning models for solving classification and regression problems.
- Our problem is a classification problem—we need to determine the category of an image.
- However, our problem is special: we’re dealing with images.
- This is why we need a special type of neural network—a convolutional neural network, which can extract visual patterns from an image and use them to make predictions.

Using a pretrained model

- For this lesson, we'll use Xception, a relatively small model that has good performance.
- First, we need to import the model itself and some helpful functions:

```
from tensorflow.keras.applications.xception import Xception  
from tensorflow.keras.applications.xception import preprocess_input  
from tensorflow.keras.applications.xception import decode_predictions
```

Using a pretrained model

- Let's load this model:

```
model = Xception(  
    weights='imagenet',  
    input_shape=(299, 299, 3)  
)
```

Using a pretrained model

- Let's test it on the image we saw previously. First, we load it using the `load_img` function:

```
img = load_img(fullname, target_size=(299, 299))
```

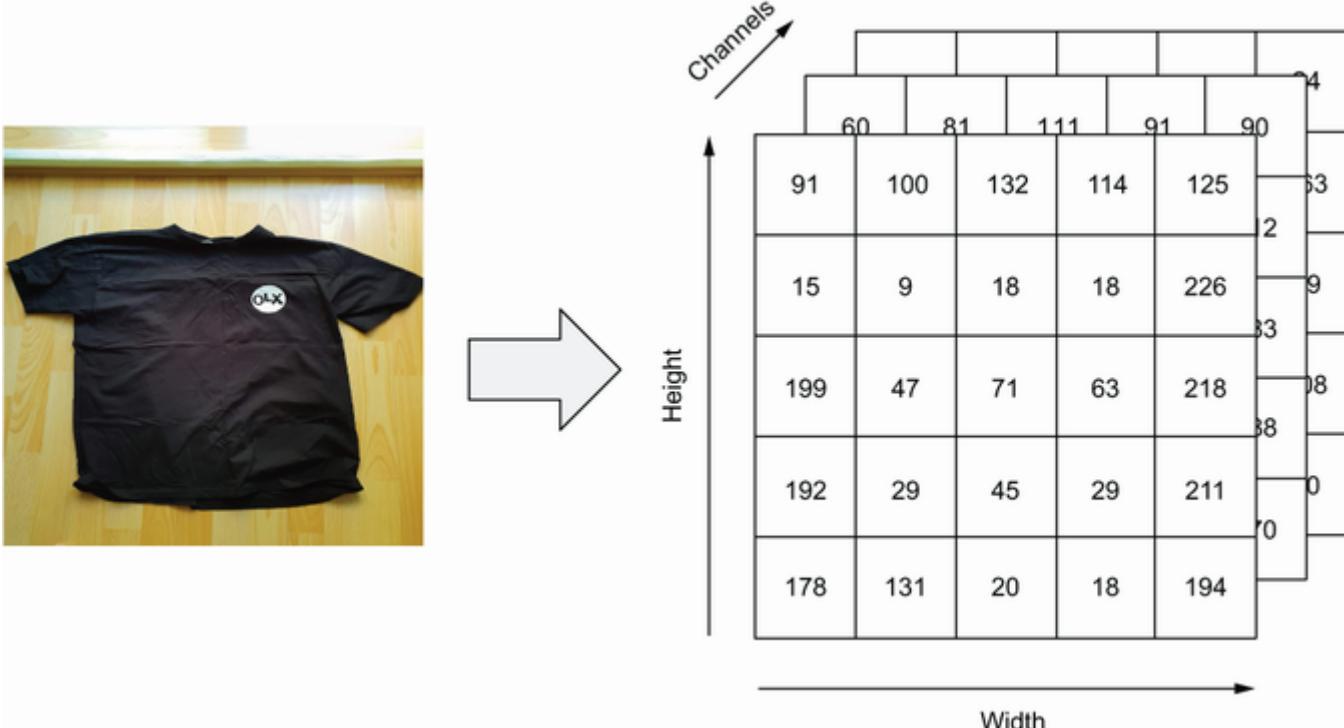
- The `img` variable is an `Image` object, which we need to convert to a NumPy array. It's easy to do:

```
x = np.array(img)
```

- This array should have the same shape as the image. Let's check it:

```
x.shape
```

Using a pretrained model



Using a pretrained model

- For example, for 10 images, the shape is (10, 299, 299, 3).
- Because we have just one image, we need to create a batch with this single image:

```
X = np.array([x])
```

- Let's check its shape:

```
X.shape
```

Using a pretrained model

- Before we can apply the model to our image, we need to prepare it with the preprocess_input function:

```
X = preprocess_input(X)
```

- This function converts the integers between 0 and 255 in the original array to numbers between –1 and 1.

Getting predictions

- To apply the model, use the predict method:

```
pred = model.predict(X)
```

- Let's take a look at this array:

```
pred.shape
```

Getting predictions

- This array is quite large—it contains 1,000 elements

```
pred = model.predict(X)
```

```
pred.shape
```

```
(1, 1000)
```

```
pred[0, :10]
```

```
array([0.0003238 , 0.00015736, 0.00021406, 0.00015296, 0.00024657,
       0.00030446, 0.00032349, 0.00014726, 0.00020487, 0.00014866],
      dtype=float32)
```

Getting predictions

- Luckily, we can use a function, decode_predictions, that decodes the prediction into meaningful class names:

`decode_predictions(pred)`

- It shows the top five most likely classes for this image:

```
[[('n02667093', 'abaya', 0.028757658),  
 ('n04418357', 'theater_curtain', 0.020734021),  
 ('n01930112', 'nematode', 0.015735716),  
 ('n03691459', 'loudspeaker', 0.013871926),  
 ('n03196217', 'digital_clock', 0.012909736)]]
```



**"Complete
Exercise"**

5. Serving models with Kubernetes and Kubeflow

Overview

This lesson covers

- Understanding different methods of deploying and serving models in the cloud
- Serving Keras and TensorFlow models with TensorFlowServing
- Deploying TensorFlow Serving to Kubernetes
- Using Kubeflow and KFServing for simplifying the deployment process

Kubernetes and Kubeflow

- Kubernetes is a container orchestration platform. It sounds complex, but it's nothing other than a place where we can deploy Docker containers.
- It takes care of exposing these containers as web services and scales these services up and down as the amount of requests we receive changes.
- Kubernetes is not the easiest tool to learn, but it's very powerful. It's likely that you will need to use it at some point, That's why we decided to cover it in this course.

Serving models with TensorFlow Serving

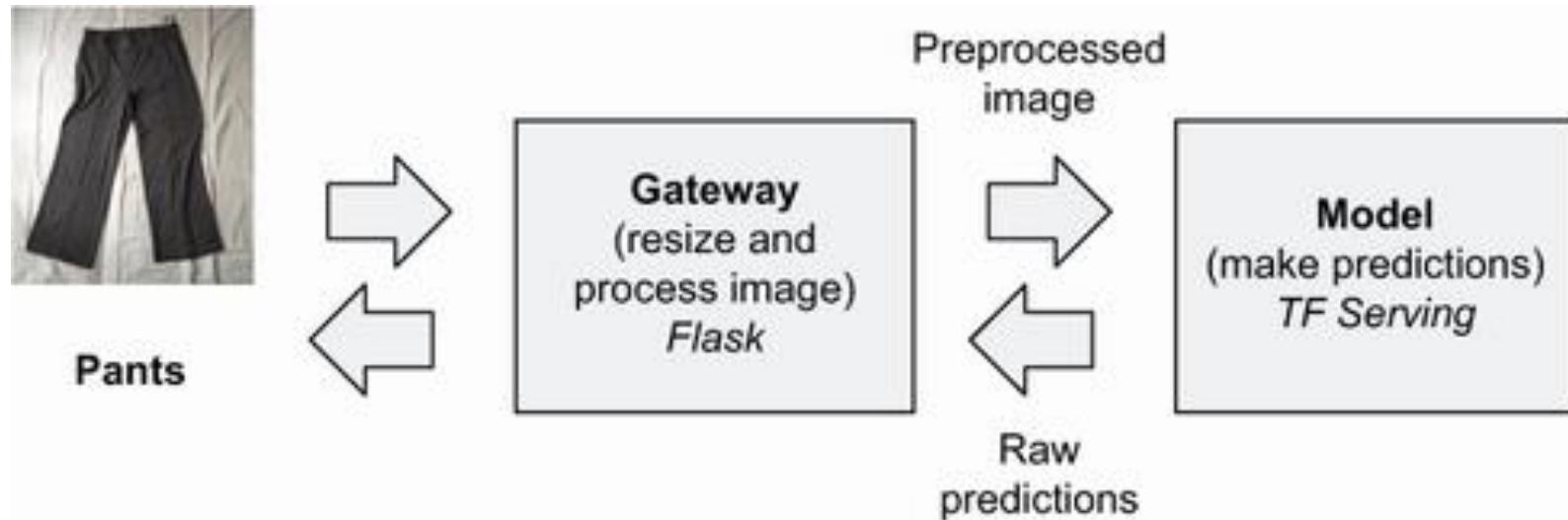
- TensorFlow Serving, usually abbreviated as “TF Serving,” is a system designed for serving TensorFlow models. Unlike TF Lite, which is made for mobile devices, TF Serving focuses on servers.
- Often, the servers have GPUs, and TF Serving knows how to make use of them.
- AWS Lambda is great for experimenting and for dealing with small amounts of images—fewer than one million per day.
- But when we grow past that amount and get more images, AWS Lambda becomes expensive.
- Then deploying models with Kubernetes and TF Serving is a better option.

Overview of the serving architecture

We need two components for a system for serving a deep learning model :

- Gateway: The preprocessing part. It gets the URL for which we need to make the prediction, prepares it, and sends it further to the model. We will use Flask for creating this service.
- Model: The part with the actual model. We will use TF Serving for this.

Overview of the serving architecture



The saved_model format

- In either case, we start with imports:

```
import tensorflow as tf  
from tensorflow import keras
```

- Then load the model:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

- And, finally, save it in the saved_model format:

```
tf.saved_model.save(model, 'clothing-model')
```

The saved_model format

- TensorFlow comes with a special utility for analyzing the models in the saved_model format—saved_model_cli.
- We don't need to install anything extra, We will use the show command from this utility:

```
saved_model_cli show --dir clothing-model --all
```

The saved_model format

- Let's take a look at the output:

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
```

```
...
```

```
signature_def['serving_default']:
```

```
  The given SavedModel SignatureDef contains the following input(s):
```

```
    inputs['input_8'] tensor_info:
```

```
      dtype: DT_FLOAT
```

```
      shape: (-1, 299, 299, 3)
```

```
      name: serving_default_input_8:0
```

```
  The given SavedModel SignatureDef contains the following output(s):
```

```
    outputs['dense_7'] tensor_info:
```

```
      dtype: DT_FLOAT
```

```
      shape: (-1, 10)
```

```
      name: StatefulPartitionedCall:0
```

```
Method name is: tensorflow/serving/predict
```

Running TensorFlow Serving locally

- All we need to do is invoke the docker run command specifying the path to the model and its name:

```
docker run -it --rm \
-p 8500:8500 \
-v "$(pwd)/clothing-model:/models/clothing-model/1" \
-e MODEL_NAME=clothing-model \
tensorflow/serving:2.3.0
```

Running TensorFlow Serving locally

- After running this command, we should see logs in the terminal:

```
2020-12-26 22:56:37.315629: I
tensorflow_serving/core/loader_harness.cc:87] Successfully loaded
servable version {name: clothing-model version: 1}
2020-12-26 22:56:37.321376: I
tensorflow_serving/model_servers/server.cc:371] Running gRPC
ModelServer at 0.0.0.0:8500 ...
[evhttp_server.cc : 238] NET_LOG: Entering the event loop ...
```

Invoking the TF Serving model from Jupyter

- Install them with pip:

```
pip install grpcio==1.32.0 tensorflow-serving-api==2.3.0
```

- We also need the keras_image_helper library for preprocessing the images.
- We already used this library in lesson 8.
- If you haven't installed it yet, use pip for that:

```
pip install keras_image_helper==0.0.1
```

Invoking the TF Serving model from Jupyter

- Next, create a Jupyter Notebook. We can call it lesson-09-image-preparation. As usual, we start with imports:

```
import grpc  
import tensorflow as tf
```

```
from tensorflow_serving.apis import predict_pb2  
from tensorflow_serving.apis import prediction_service_pb2_grpc
```

Invoking the TF Serving model from Jupyter

- Now we need to define the connection to our service:

```
host = 'localhost:8500'  
channel = grpc.insecure_channel(host)  
stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```

Invoking the TF Serving model from Jupyter

- For preprocessing the images, we use the keras_image_helper library, like previously:

```
from keras_image_helper import create_preprocessor  
  
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Invoking the TF Serving model from Jupyter

- Let's convert it to a NumPy array:

```
url = "http://bit.ly/mlbookcamp-pants"  
X = preprocessor.from_url(url)
```



Invoking the TF Serving model from Jupyter

- We have a NumPy array in X, but we can't use it as is. For gRPC, we need to convert it to protobuf.
- TensorFlow has a special function for that: `tf.make_tensor_proto`.
- This is how we use it:

```
def np_to_protobuf(data):
    return tf.make_tensor_proto(data, shape=data.shape)
```

Invoking the TF Serving model from Jupyter

- Now we can use the np_to_protobuf function to prepare a gRPC request:

```
pb_request = predict_pb2.PredictRequest()
```

```
pb_request.model_spec.name = 'clothing-model'
```

```
pb_request.model_spec.signature_name = 'serving_default'
```

```
pb_request.inputs['input_8'].CopyFrom(np_to_protobuf(X))
```

Invoking the TF Serving model from Jupyter

- Let's execute it:

```
pb_result = stub.Predict(pb_request, timeout=20.0)
```

- This sends a request to the TF Serving instance, Then TF Serving applies the model to the request and sends back the results.
- The results are saved to the pb_result variable. To get the predictions from there, we need to access one of the outputs:

```
pred = pb_result.outputs['dense_7'].float_val
```

Invoking the TF Serving model from Jupyter

- The pred variable is a list of floats—the predictions:

```
[-1.868, -4.761, -2.316, -1.062, 9.887, -2.812, -3.666, 3.200, -2.602, -4.835]
```

```
labels = [  
    'dress',  
    'hat',  
    'longsleeve',  
    'outwear',  
    'pants',  
    'shirt',  
    'shoes',  
    'shorts',  
    'skirt',  
    't-shirt'  
]
```

```
result = {c: p for c, p in zip(labels, pred)}
```

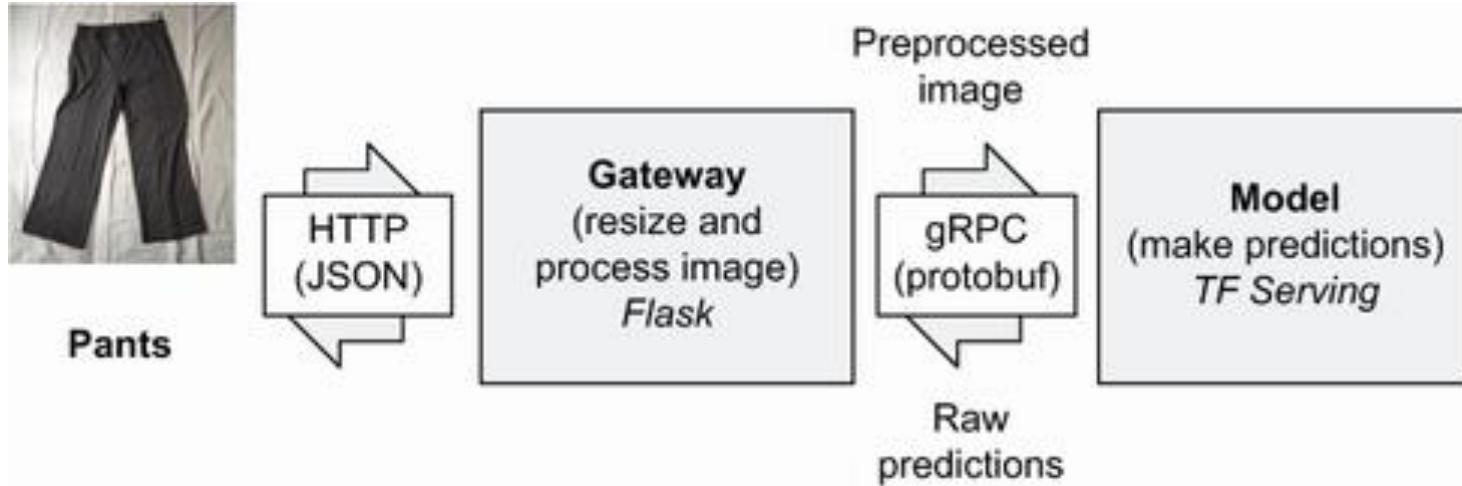
Invoking the TF Serving model from Jupyter

Invoking the TF Serving model from Jupyter

- This gives us the final result:

```
{'dress': -1.868,  
 'hat': -4.761,  
 'longsleeve': -2.316,  
 'outwear': -1.062,  
 'pants': 9.887,  
 'shirt': -2.812,  
 'shoes': -3.666,  
 'shorts': 3.200,  
 'skirt': -2.602,  
 't-shirt': -4.835}
```

Creating the Gateway service



Creating the Gateway service

- First, we get the same imports as we have in the notebook:

```
import grpc
import tensorflow as tf
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc

from keras_image_helper import create_preprocessor
```

Creating the Gateway service

- Now we need to add Flask imports:

```
from flask import Flask, request, jsonify
```

- Next, create the connection gRPC stub:

```
host = os.getenv('TF_SERVING_HOST', 'localhost:8500')
channel = grpc.insecure_channel(host)
stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
```

Creating the Gateway service

- Now let's create the preprocessor:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

- Also, we need to define the names of our classes:

```
labels = [  
    'dress',  
    'hat',  
    'longsleeve',  
    'outwear',  
    'pants',  
    'shirt',  
    'shoes',  
    'shorts',  
    'skirt',  
    't-shirt'  
]
```

Creating the Gateway service

- Let's start with make_request:

```
def np_to_protobuf(data):  
    return tf.make_tensor_proto(data, shape=data.shape)  
  
def make_request(X):  
    pb_request = predict_pb2.PredictRequest()  
    pb_request.model_spec.name = 'clothing-model'  
    pb_request.model_spec.signature_name = 'serving_default'  
    pb_request.inputs['input_8'].CopyFrom(np_to_protobuf(X))  
    return pb_request
```

Creating the Gateway service

- Next, create process_response:

```
def process_response(pb_result):
    pred = pb_result.outputs['dense_7'].float_val
    result = {c: p for c, p in zip(labels, pred)}
    return result
```

- And finally, let's put everything together:

```
def apply_model(url):
    X = preprocessor.from_url(url)
    pb_request = make_request(X)
    pb_result = stub.Predict(pb_request, timeout=20.0)
    return process_response(pb_result)
```

Creating the Gateway service

- All the code is ready. We only need to do one last thing: create a Flask app and the predict function. Let's do it:

```
app = Flask('clothing-model')
```

```
@app.route('/predict', methods=['POST'])
```

```
def predict():
```

```
    url = request.get_json()
```

```
    result = apply_model(url['url'])
```

```
    return jsonify(result)
```

```
if __name__ == "__main__":
```

```
    app.run(debug=True, host='0.0.0.0', port=9696)
```

Creating the Gateway service

- Now we're ready to run the service. Execute this command in the terminal:

```
python model_server.py
```

- Wait until it's ready. We should see the following in the terminal:
* Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)

Creating the Gateway service

- We need to send a request with a URL and show the response. This is how we do it with requests:

```
import requests
```

```
req = {  
    "url": "http://bit.ly/mlbookcamp-pants"  
}
```

```
url = 'http://localhost:9696/predict'
```

```
response = requests.post(url, json=req)  
response.json()
```

Creating the Gateway service

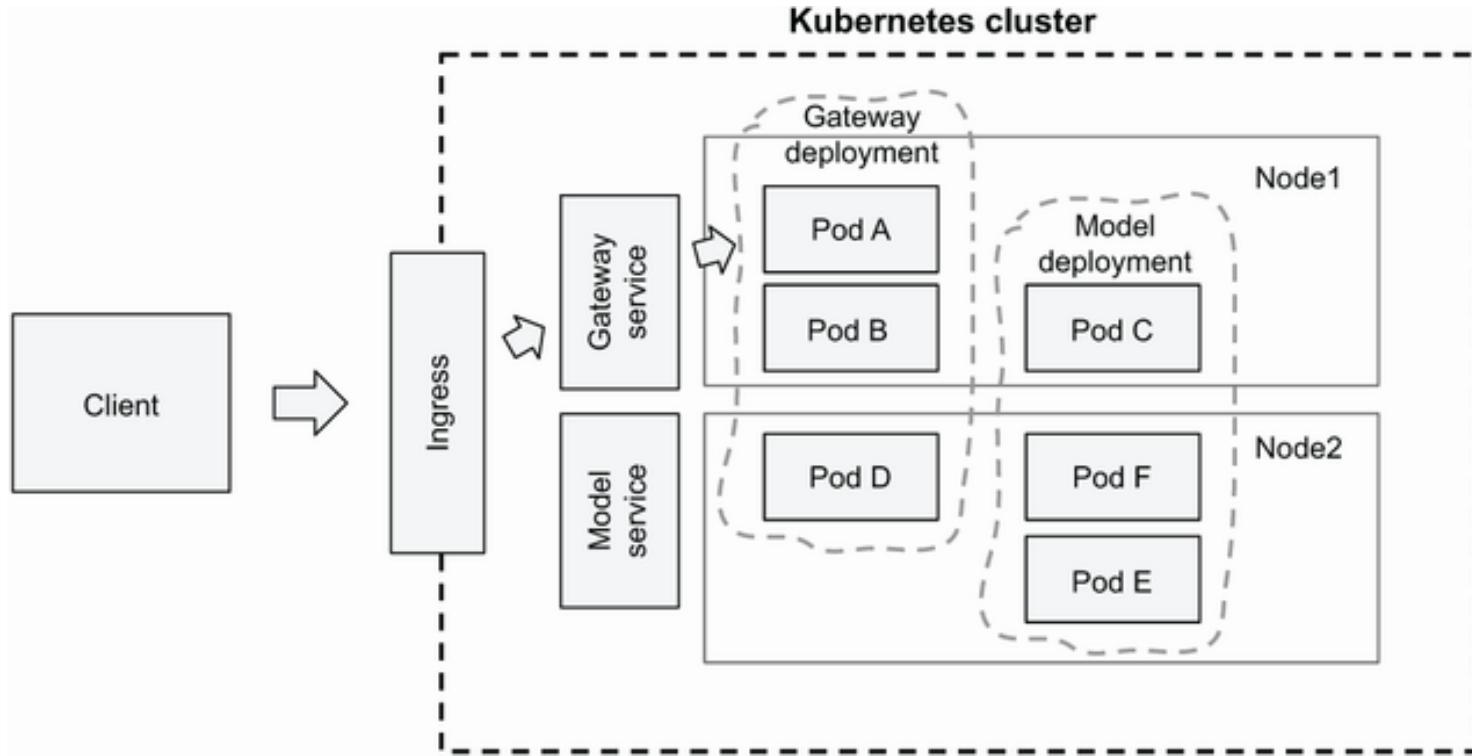
- We send a POST request to our service and display the results. The response is the same as previously:

```
{'dress': -1.868,  
 'hat': -4.761,  
 'longsleeve': -2.316,  
 'outwear': -1.062,  
 'pants': 9.887,  
 'shirt': -2.812,  
 'shoes': -3.666,  
 'shorts': 3.200,  
 'skirt': -2.602,  
 't-shirt': -4.835}
```

Introduction to Kubernetes

- The main unit of abstraction in Kubernetes is a pod. A pod contains a single Docker image, and when we want to serve something, pods do the actual job.
- Pods live on a node—this is an actual machine, A node usually contains one or more pods.
- To deploy an application, we define a deployment, We specify how many pods the application should have and which image should be used.

Introduction to Kubernetes



Creating a Kubernetes cluster on AWS

- First, prepare a file with the cluster configuration. Create a file in your project directory and call it cluster.yaml:

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: ml-bookcamp-eks
  region: eu-west-1
  version: "1.18"
nodeGroups:
  - name: ng
    desiredCapacity: 2
    instanceType: m5.xlarge
```

Creating a Kubernetes cluster on AWS

- After creating the config file, we can use eksctl for spinning up a cluster:

```
eksctl create cluster -f cluster.yaml
```

- Once it's created, we need to configure kubectl to be able to access it. For AWS, we do this with the AWS CLI:

```
aws eks --region eu-west-1 update-kubeconfig --name ml-bookcamp-eks
```

Creating a Kubernetes cluster on AWS

- Now let's check that everything works, and that we can connect to our cluster using kubectl:

`kubectl get service`

- This command returns the list of currently running services.
- We haven't deployed anything, so we expect to see only one service—Kubernetes itself.
- This is the result you should see:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.100.0.1	<none>	443/TCP	6m17s

Preparing the Docker images

- we first need to publish our image to ECR—the Docker registry of AWS.
- Let's create a registry called model-serving:

```
aws ecr create-repository --repository-name model-serving
```

- It should return a path like this:

```
<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/model-serving
```

Preparing the Docker images

- When running a Docker image of TF Serving locally, we used this command (you don't need to run it now):

```
docker run -it --rm \
-p 8500:8500 \
-v "$(pwd)/clothing-model:/models/clothing-model/1" \
-e MODEL_NAME=clothing-model \
tensorflow/serving:2.3.0
```

Preparing the Docker images

- Let's create a Dockerfile for that. We can name it `tf-serving.dockerfile`:

```
FROM tensorflow/serving:2.3.0
```

```
ENV MODEL_NAME clothing-model  
COPY clothing-model /models/clothing-model/1
```

Preparing the Docker images

- Let's build it:

```
IMAGE_SERVING_LOCAL="tf-serving-clothing-model"  
docker build -t ${IMAGE_SERVING_LOCAL} -f tf-serving.dockerfile .
```

- Next, we need to publish this image to ECR. First, we need to authenticate with ECR using AWS CLI:

```
$(aws ecr get-login --no-include-email)
```

Preparing the Docker images

- Next, tag the image with the remote URI:

```
ACCOUNT=XXXXXXXXXXXXXX
```

```
REGION=eu-west-1
```

```
REGISTRY=${ACCOUNT}.dkr.ecr.${REGION}.amazonaws.com/model-serving
```

```
IMAGE_SERVING_REMOTE=${REGISTRY}:${IMAGE_SERVING_LOCAL}
```

```
docker tag ${IMAGE_SERVING_LOCAL} ${IMAGE_SERVING_REMOTE}
```

- Be sure to change the ACCOUNT and REGION variables.
- Now we're ready to push the image to ECR:

```
docker push ${IMAGE_SERVING_REMOTE}
```

Preparing the Docker images

- Remember that in lesson 5, we used Pipenv for managing dependencies.
- Let's use it here as well:

```
pipenv install flask gunicorn \
    keras_image_helper==0.0.1 \
    grpcio==1.32.0 \
    tensorflow==2.3.0 \
    tensorflow-serving-api==2.3.0
```

Preparing the Docker images

```
FROM python:3.7.5-slim
ENV PYTHONUNBUFFERED=TRUE
RUN pip --no-cache-dir install pipenv

WORKDIR /app

COPY ["Pipfile", "Pipfile.lock", "./"]
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache
COPY "model_server.py" "model_server.py"

EXPOSE 9696

ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "model_server:app"]
```

Preparing the Docker images

- Let's build this image now:

```
IMAGE_GATEWAY_LOCAL="serving-gateway"
```

```
docker build -t ${IMAGE_GATEWAY_LOCAL} -f gateway.dockerfile
```

- And push it to ECR:

```
IMAGE_GATEWAY_REMOTE=${REGISTRY}:${IMAGE_GATEWAY_LOCAL}
```

```
docker tag ${IMAGE_GATEWAY_LOCAL} ${IMAGE_GATEWAY_REMOTE}
```

```
docker push ${IMAGE_GATEWAY_REMOTE}
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tf-serving-clothing-model
  labels:
    app: tf-serving-clothing-model
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tf-serving-clothing-model
  template:
    metadata:
      labels:
        app: tf-serving-clothing-model
    spec:
      containers:
        - name: tf-serving-clothing-model
          image: <ACCOUNT>.dkr.ecr.<REGION>.
            ➔amazonaws.com/model-serving:tf-serving-clothing-model
      ports:
        - containerPort: 8500
```

Deploying to Kubernetes

Deploying to Kubernetes

- We have a config. Now we need to use it to create a Kubernetes object—a deployment in our case.
- We do it by using the `apply` command from `kubectl`:

```
kubectl apply -f tf-serving-clothing-model-deployment.yaml
```

Deploying to Kubernetes

- To verify that it's working, we need to check if a new deployment appeared.
- This is how we can get the list of all active deployments:

`kubectl get deployments`

- The output should look similar to this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
tf-serving-clothing-model	1/1	1	1	41s

Deploying to Kubernetes

- We see that our deployment is there.
- Also, we can get the list of pods. It's quite similar to getting the list of all deployments:

`kubectl get pods`

- We should see something like that in the output:

NAME	READY	STATUS	RESTARTS	AGE
tf-serving-clothing-model-56bc84678d-b6n4r	1/1	Running	0	108s

```
apiVersion: v1
kind: Service
metadata:
  name: tf-serving-clothing-model
  labels:
    app: tf-serving-clothing-model
spec:
  ports:
    - port: 8500
      targetPort: 8500
      protocol: TCP
      name: http
  selector:
    app: tf-serving-clothing-model
```

Deploying to Kubernetes

Deploying to Kubernetes

- We apply it in the same way—by using the apply command:

```
kubectl apply -f tf-serving-clothing-model-service.yaml
```

- To check that it works, we can get the list of all services and see if our service is there:

```
kubectl get services
```

Deploying to Kubernetes

- We should see something like

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.100.0.1	<none>	443/TCP	84m
tf-serving-clothing-model	ClusterIP	10.100.111.165	<none>	8500/TCP	19s

Deploying to Kubernetes

- To access this service, we need to get its URL. The internal URLs typically follow this pattern:

<service-name>.<namespace-name>.svc.cluster.local

- This is the URL for the service we just created:

tf-serving-clothing-model.default.svc.cluster.local

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: serving-gateway
  labels:
    app: serving-gateway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: serving-gateway
  template:
    metadata:
      labels:
        app: serving-gateway
    spec:
      containers:
        - name: serving-gateway
          image: <ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/model-serving:serving-gateway
          ports:
            - containerPort: 9696
      env:
        - name: TF_SERVING_HOST
          value: "tf-serving-clothing-model.default.svc.cluster.local:8500"
```

Deploying to Kubernetes

1

Deploying to Kubernetes

- Let's apply this configuration:

```
kubectl apply -f serving-gateway-deployment.yaml
```

- This should create a new pod and a new deployment. Let's take a look at the list of pods:

```
kubectl get pod
```

Deploying to Kubernetes

- Indeed, a new pod is there:

NAME	READY	STATUS	RESTARTS	AGE
tf-serving-clothing-model-56bc84678d-b6n4r	1/1	Running	0	1h
serving-gateway-5f84d67b59-lx8tq	1/1	Running	0	30s

```
apiVersion: v1
kind: Service
metadata:
  name: serving-gateway
  labels:
    app: serving-gateway
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 9696
      protocol: TCP
      name: http
  selector:
    app: serving-gateway
```

Deploying to Kubernetes

Deploying to Kubernetes

- Let's apply this config:

```
kubectl apply -f serving-gateway-service.yaml
```

- To see the external URL of the service, use the describe command:

```
kubectl describe service serving-gateway
```

Name: serving-gateway
Namespace: default
Labels: <none>
Annotations: <none>
Selector: app=serving-gateway
Type: LoadBalancer
IP Families: <none>
IP: 10.100.100.24
IPs: <none>

LoadBalancer Ingress: ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-1.elb.amazonaws.com

Port: http 80/TCP
TargetPort: 9696/TCP
NodePort: http 32196/TCP
Endpoints: <none>

Session Affinity: None
External Traffic Policy: Cluster
Events:

Deploying to Kubernetes

- We're interested in the line with LoadBalancer Ingress.
- This is the URL we need to use to access the Gateway service.
- In our case, this is the URL:

ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-1.elb.amazonaws.com

Testing the service

```
import requests
```

```
req = {  
    "url": "http://bit.ly/mlbookcamp-pants"  
}
```

```
url = 'http://ad1fad0c1302141989ed8ee449332e39-117019527.eu-west-  
1.elb.amazonaws.com/predict'
```

```
response = requests.post(url, json=req)  
response.json()
```

Testing the service

- Run it. As a result, we get the same predictions as previously:

```
{"dress": -1.86829,  
'hat': -4.76124,  
'longsleeve': -2.31698,  
'outwear': -1.06257,  
'pants': 9.88716,  
'shirt': -2.81243,  
'shoes': -3.66628,  
'shorts': 3.20036,  
'skirt': -2.60233,  
't-shirt': -4.83504}
```

Model deployment with Kubeflow

Kubeflow is a project that aims to simplify the deployment of machine learning services on Kubernetes.

It consists of a set of tools, each of which aims at solving a particular problem. For example:

- Kubeflow Notebooks Server: Makes it easier to centrally host Jupyter Notebooks
- Kubeflow Pipelines: Automates the training process
- Katib: Selects the best parameters for the model
- Kubeflow Serving (abbreviated as “KFServing”): Deploys machine learning models

the model: Uploading it to S3

- We can create it with the AWS CLI:

```
aws s3api create-bucket \  
  --bucket mlbookcamp-models-alexey \  
  --region eu-west-1 \  
  --create-bucket-configuration LocationConstraint=eu-west-1
```

- After creating a bucket, we need to upload the model there. Use the AWS CLI for that:

```
aws s3 cp --recursive clothing-model s3:/mlbookcamp-models-  
  alexey/clothing-model/0001/
```

Deploying TensorFlow models with KFServing

- First, create another YAML file (tf-clothes.yaml) with the following content:

```
apiVersion: "serving.kubeflow.org/v1beta1"
kind: "InferenceService"
metadata:
  name: "clothing-model"
spec:
  default:
    predictor:
      serviceAccountName: sa
    tensorflow:
      storageUri: "s3:/mlbookcamp-models-alexey/clothing-model"
```

Deploying TensorFlow models with KFServing

- Like with usual Kubernetes, we use kubectl to apply this config:

```
kubectl apply -f tf-clothing.yaml
```

- Because it creates an InferenceService object, we need to get the list of such objects using the get command from kubectl:

```
kubectl get inferenceservice
```

Deploying TensorFlow models with KFServing

- We should see something like this:

NAME	URL	READY	AGE
clothing-model	http://clothing-model...	True	... 97s

- If our service READY is not yet True, we need to wait a bit before it becomes ready. It may take 1–2 minutes.

Accessing the model

- The model is deployed. Let's use it! For that, we can start a Jupyter Notebook or create a Python script file.
- KFServing uses HTTP and JSON, so we use the requests library for communicating with it. So let's start by importing it:

```
import requests
```

Accessing the model

- we need to use the image preprocessor for preparing the images. It's the same one we used previously:

```
from keras_image_helper import create_preprocessor  
  
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Accessing the model

- Now, we need an image for testing.
- We use the same image of pants as in the previous section and use the same code for getting it and preprocessing it:

```
image_url = "http://bit.ly/mlbookcamp-pants"
```

```
X = preprocessor.from_url(image_url)
```

- The X variable contains a NumPy array. We need to convert it to a list before we can send the data to KFServing:

```
data = {  
    "instances": X.tolist()  
}
```

Accessing the model

- With these changes, this is how the URL appears:

```
url = 'https://clothing-
model.default.kubeflow.mlbookcamp.com/v1/models/clothing-
model:predict'
```

- We're ready to post the request:

```
resp = requests.post(url, json=data)
results = resp.json()
```

Accessing the model

- Let's take a look at the results:

```
{'predictions': [[-1.86828923,  
-4.76124525,  
-2.31698346,  
-1.06257045,  
9.88715553,  
-2.81243205,  
-3.66628242,  
3.20036,  
-2.60233665,  
-4.83504581]]}
```

Accessing the model

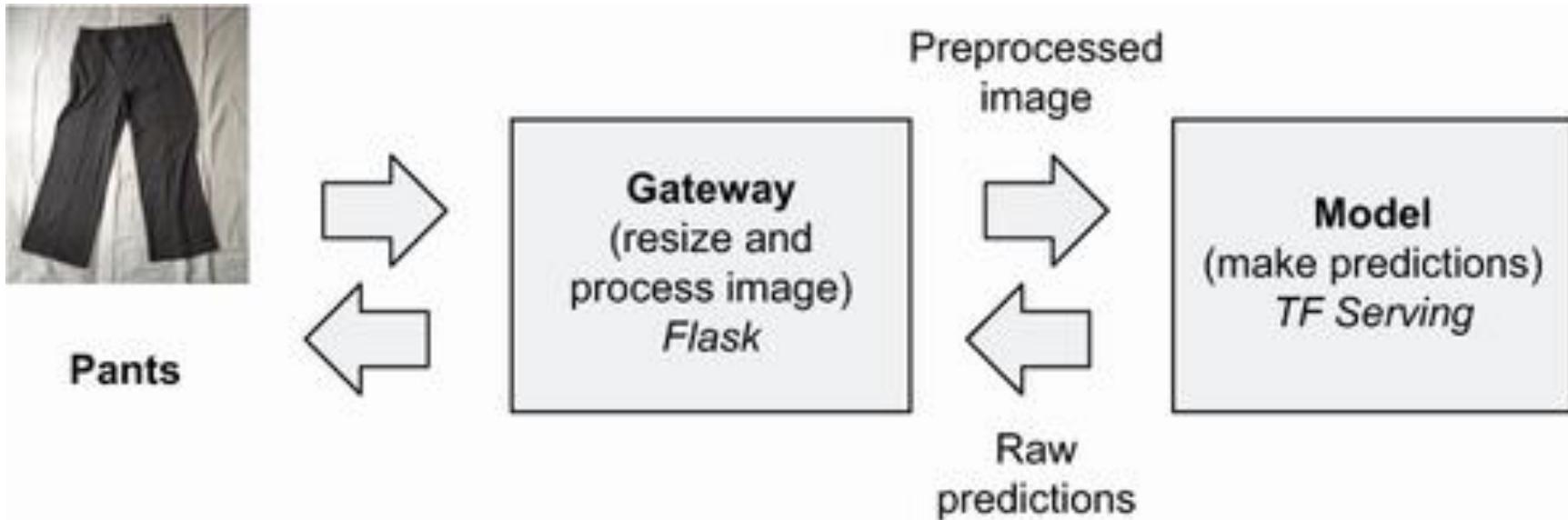
```
pred = results['predictions'][0]
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]
result = {c: p for c, p in zip(labels, pred)}
```

Accessing the model

- Here's the result:

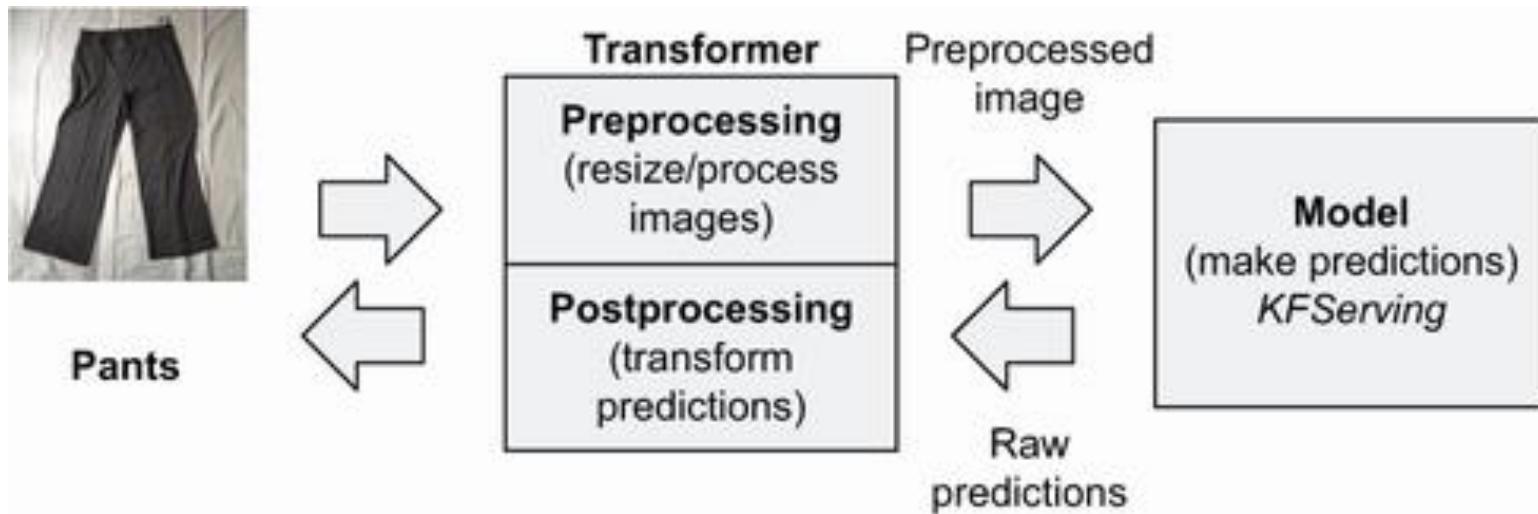
```
{'dress': -1.86828923,  
'hat': -4.76124525,  
'longsleeve': -2.31698346,  
'outwear': -1.06257045,  
'pants': 9.88715553,  
'shirt': -2.81243205,  
'shoes': -3.66628242,  
'shorts': 3.20036,  
'skirt': -2.60233665,  
't-shirt': -4.83504581}
```

KFServing transformers



KFServing transformers

- We can put all the preprocessing code from the previous section into a transformer



KFServing transformers

- It looks like this:

```
class ImageTransformer(kfserving.KFModel):  
    def preprocess(self, inputs):  
        # implement pre-processing logic  
  
    def postprocess(self, inputs):  
        # implement post-processing logic
```

KFServing transformers

- Let's use it. First, we need to delete the old inference service:

```
kubectl delete -f tf-clothes.yaml
```

KFServing transformers

```
apiVersion: "serving.kubeflow.org/v1alpha2"
kind: "InferenceService"
metadata:
  name: "clothing-model"
spec:
  default:
    predictor:
      serviceAccountName: sa
      tensorflow:
        storageUri: "s3://mlbookcamp-models-alexey/clothing-model"
  transformer:
    custom:
      container:
        image: "agrigorev/kfserving-keras-transformer:0.0.1"
        name: user-container
        env:
          - name: MODEL_INPUT_SIZE
            value: "299,299"
          - name: KERAS_MODEL_NAME
            value: "xception"
          - name: MODEL_LABELS
            value: "dress,hat,longsleeve,outwear,pants,
ehttps://shirt,shoes,shorts,skirt,t-shirt"
```

1

2

3

4

KFServing transformers

- Let's apply this config:

```
kubectl apply -f tf-clothes.yaml
```

Testing the transformer

- This is how it looks:

```
import requests
```

```
data = {  
    "instances": [  
        {"url": "http://bit.ly/mlbookcamp-pants"},  
    ]  
}  
  
url = 'https://clothing-  
model.default.kubeflow.mlbookcamp.com/v1/models/clothing-  
model:predict'  
  
result = requests.post(url, json=data).json()
```

Testing the transformer

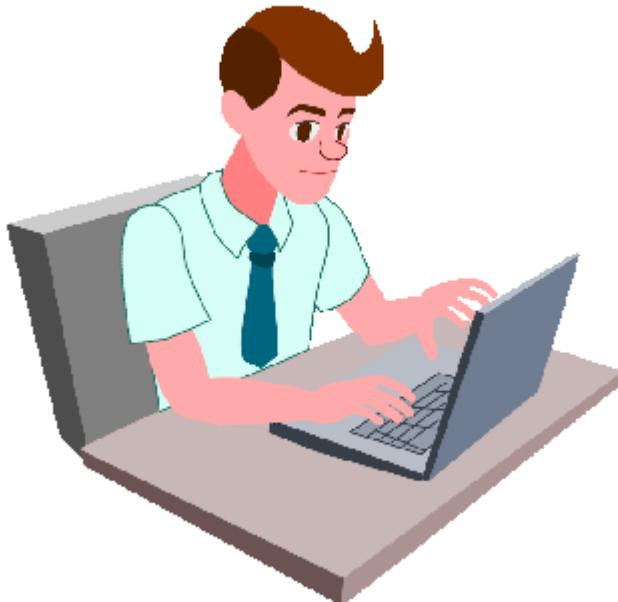
- The URL of the service stays the same.
- The result contains the predictions:

```
{"predictions": [{"dress": -1.8682, "hat": -4.7612, "longsleeve": -2.3169, "outwear": -1.0625, "pants": 9.8871, "shirt": -2.8124, "shoes": -3.6662, "shorts": 3.2003, "skirt": -2.6023, "t-shirt": -4.8350}]} 
```

Deleting the EKS cluster

- After experimenting with EKS, don't forget to shut down the cluster.
Use eksctl for that:

```
eksctl delete cluster --name ml-bookcamp-eks
```



"Complete Exercises"

Summary

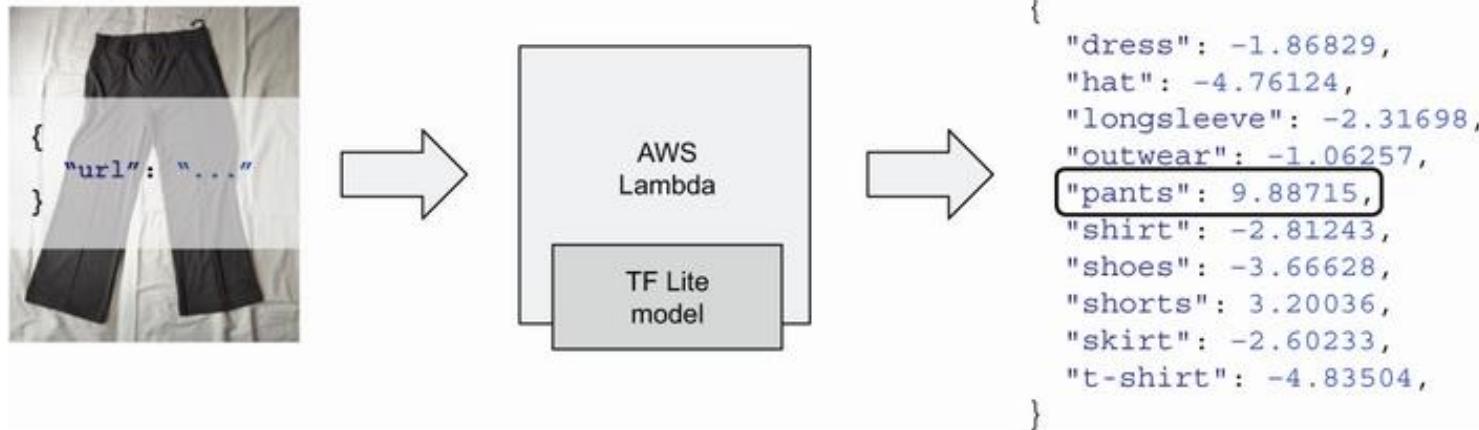
- TensorFlow-Serving is a system for deploying Keras and TensorFlow models.
- It uses gRPC and protobuf for communication, and it's highly optimized for serving.
- When using TensorFlow Serving, we typically need a component for preparing the user request into the format the model expects.

6. Serverless Deep Learning

Serverless: AWS Lambda



We use AWS Lambda for deploying the model. For doing that, we'll also use TensorFlow Lite—a lightweight version of TensorFlow that has only the most essential functions.



Serverless: AWS Lambda

We want to build a web service that

- Gets the URL in the request
- Loads the image from this URL
- Uses TensorFlow Lite to apply the model to the image and get the predictions
- Responds with the results

To create this service, we will need to

- Convert the model from Keras to the TensorFlow Lite format
- Preprocess the images—resize them and apply the preprocessing function
- Package the code in a Docker image, and upload it to ECR (the Docker registry from AWS)
- Create and test the lambda function on AWS
- Make the lambda function available to everyone with AWS API Gateway

TensorFlow Lite



TensorFlow

- TensorFlow is a great framework with a rich set of features. However, most of these features are not needed for model deployment, and they take up a lot of space: when compressed, TensorFlow takes up more than 1.5 GB of space.

Let's install the library now. We can do so with pip:

```
pip install --extra-index-url https://google-coral.github.io/py-  
repo/tflite_runtime
```

Converting the model to TF Lite format

- We need to convert our model to TF-Lite format.

```
wget https://github.com/fenago/mlbookcamp-code/releases/download/  
lesson7-model/xception_v4_large_08_0.894.h5
```

- Now let's create a simple script for converting this model—convert.py.

First, start with imports:

```
import tensorflow as tf  
from tensorflow import keras
```

Converting the model to TF Lite format

- Next, load the Keras model:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

- And finally, convert it to TF Lite:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
```

```
tflite_model = converter.convert()
```

```
with tf.io.gfile.GFile('clothing-model-v4.tflite', 'wb') as f:
```

```
    f.write(tflite_model)
```

Converting the model to TF Lite format

- Let's run it:

```
python convert.py
```

10000110	0
11100100	0
00010000	1
00000100	0
00010001	1
10100000	1
11000011	1
11101111	0
10110001	1
11100000	1

Preparing the images

- When testing the model in Keras, we preprocessed each image using the preprocess_input function. This is how we imported it previously:

```
from tensorflow.keras.applications.xception import preprocess_input
```

And then we applied this function to images before we put them into models.

Preparing the images

- Let's install it with pip:

```
pip install keras_image_helper
```

Next, open Jupyter, and create a notebook called lesson-08-model-test.

- We start by importing the create_preprocessor function from the library:

```
from keras_image_helper import create_preprocessor
```

Preparing the images

- We used the Xception model, and it expects an image of size 299 × 299. Let's create a preprocessor for our model:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

- Now let's get a picture of pants (figure), and prepare it:

```
image_url = 'http://bit.ly/mlbookcamp-pants'
```

```
X = preprocessor.from_url(image_url)
```



Using the TensorFlow Lite model



- We have the array X from the previous step, and now we can use TF Lite for classifying it.

First, import TF Lite:

```
import tensorflow as tf
```

- Load the model we previously converted:

```
interpreter = tf.lite.Interpreter(model_path='clothing-model-v4.tflite')
interpreter.allocate_tensors()
```

Using the TensorFlow Lite model

- To be able to use the model, we need to get its input (where X will go) and the output (where we get the predictions from):

```
input_details = interpreter.get_input_details()
```

```
input_index = input_details[0]['index']
```

```
output_details = interpreter.get_output_details()
```

```
output_index = output_details[0]['index']
```

Using the TensorFlow Lite model

- To apply the model, take the X we previously prepared, put it into the input, invoke the interpreter, and get the results from the output:

```
interpreter.set_tensor(input_index, X)
```

```
interpreter.invoke()
```

```
preds = interpreter.get_tensor(output_index)
```

- The preds array contains the predictions:

```
array([[-1.8682897, -4.7612453, -2.316984 , -1.0625705, 9.887156 ,  
       -2.8124316, -3.6662838, 3.2003622, -2.6023388, -4.8350453]],  
      dtype=float32)
```

Using the TensorFlow Lite model

- Now we can do the same thing with it as previously—assign the label to each element of this array:

```
labels = [  
    'dress',  
    'hat',  
    'longsleeve',  
    'outwear',  
    'pants',  
    'shirt',  
    'shoes',  
    'shorts',  
    'skirt',  
    't-shirt'  
]
```

Using the TensorFlow Lite model

- It's done! We have the predictions in the results variable:

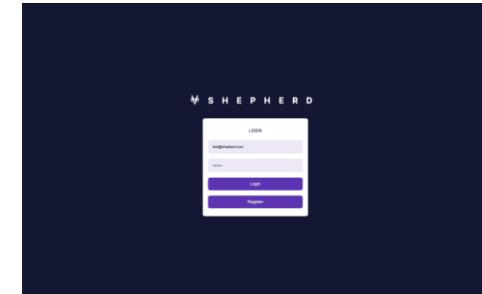
```
{'dress': -1.8682897,  
 'hat': -4.7612453,  
 'longsleeve': -2.316984,  
 'outwear': -1.0625705,  
 'pants': 9.887156,  
 'shirt': -2.8124316,  
 'shoes': -3.6662838,  
 'shorts': 3.2003622,  
 'skirt': -2.6023388,  
 't-shirt': -4.8350453}
```

Code for the lambda function

- In the previous section, we wrote all the code we need for the lambda function. Let's put it together in a single script—lambda_function.py.

As usual, start with imports:

```
import tensorflow.lite_runtime.interpreter as tflite  
from keras_image_helper import create_preprocessor
```



- Then, create the preprocessor:

```
preprocessor = create_preprocessor('xception', target_size=(299, 299))
```

Code for the lambda function

- Next, load the model, and get the output and input:

```
interpreter = tflite.Interpreter(model_path='clothing-model-v4.tflite')
```

```
interpreter.allocate_tensors()
```

```
input_details = interpreter.get_input_details()
```

```
input_index = input_details[0]['index']
```

```
output_details = interpreter.get_output_details()
```

```
output_index = output_details[0]['index']
```

Code for the lambda function

- To make it a bit cleaner, we can put all the code for making a prediction together in one function:

```
def predict(X):  
    interpreter.set_tensor(input_index, X)  
    interpreter.invoke()  
    preds = interpreter.get_tensor(output_index)  
    return preds[0]
```

Code for the lambda function

- Next, let's make another function for preparing the results:

```
labels = [  
    'dress',  
    'hat',  
    'longsleeve',  
    'outwear',  
    'pants',  
    'shirt',  
    'shoes',  
    'shorts',  
    'skirt',  
    't-shirt'  
]  
  
def decode_predictions(pred):  
    result = {c: float(p) for c, p in zip(labels, pred)}  
    return result
```

Code for the lambda function

- Finally, put everything together in one function—lambda_handler—which is the function invoked by the AWS Lambda environment. It will use all the things we defined previously:

```
def lambda_handler(event, context):  
    url = event['url']  
    X = preprocessor.from_url(url)  
    preds = predict(X)  
    results = decode_predictions(preds)  
    return results
```



Code for the lambda function

```
{  
    "url": "http://bit.ly/mlbookcamp-pants"  
}
```

```
def lambda_handler(event, context):  
    url = event['url']  
    X = preprocessor.from_url(url)  
    preds = predict(X)  
    results = decode_predictions(preds)  
    return results
```

```
{  
    "dress": -1.86829,  
    "hat": -4.76124,  
    "longsleeve": -2.31698,  
    "outwear": -1.06257,  
    "pants": 9.88715,  
    "shirt": -2.81243,  
    "shoes": -3.66628,  
    "shorts": 3.20036,  
    "skirt": -2.60233,  
    "t-shirt": -4.83504,  
}
```

Preparing the Docker image



- First, create a file named Dockerfile:

```
FROM public.ecr.aws/lambda/python:3.7
```

```
RUN pip3 install keras_image_helper --no-cache-dir
```

```
RUN pip3 install https://raw.githubusercontent.com/fenago/serverless-deep-  
learning
```

```
➥/master/tflite/tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl
```

```
➥--no-cache-dir
```

```
COPY clothing-model-v4.tflite clothing-model-v4.tflite
```

```
COPY lambda_function.py lambda_function.py
```

```
CMD [ "lambda_function.lambda_handler" ]
```

Preparing the Docker image

- Let's build this image:

```
docker build -t tf-lite-lambda .
```

- Next, we need to check that the lambda function works. Let's run the image:

```
docker run --rm -p 8080:8080 tf-lite-lambda
```

Preparing the Docker image

- We can continue using the Jupyter Notebook we created earlier, or we can create a separate Python file named test.py. It should have the following content:

```
import requests
```

```
data = {  
    "url": "http://bit.ly/mlbookcamp-pants"  
}
```

```
url = "http://localhost:8080/2015-03-31/functions/function/invocations"
```

```
results = requests.post(url, json=data).json()  
print(results)
```

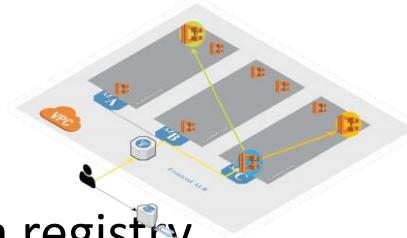
Preparing the Docker image

- When we run it, we get the following response:

```
{  
  "dress": -1.86829,  
  "hat": -4.76124,  
  "longsleeve": -2.31698,  
  "outwear": -1.06257,  
  "pants": 9.88715,  
  "shirt": -2.81243,  
  "shoes": -3.66628,  
  "shorts": 3.20036,  
  "skirt": -2.60233,  
  "t-shirt": -4.83504  
}
```



Pushing the image to AWS ECR



- To publish this Docker image to AWS, we first need to create a registry using the AWS CLI tool:

```
aws ecr create-repository --repository-name lambda-images
```

- It will return back an URL that looks like this:

```
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/lambda-images
```

Pushing the image to AWS ECR

- Once the registry is created, we need to push the image there. Because this registry belongs to our account, we first need to authenticate our Docker client. On Linux and MacOS, you can do this:

```
$(aws ecr get-login --no-include-email)
```

On Windows, *run aws ecr get-login --no-include-email*, copy the output, enter it into the terminal, and execute it manually.

Pushing the image to AWS ECR

- Now let's use the registry URL to push the image to ECR:

```
REGION=eu-west-1
```

```
ACCOUNT=XXXXXXXXXXXXXX
```

```
REMOTE_NAME=${ACCOUNT}.dkr.ecr.${REGION}.amazonaws.com/lambda-  
images:tf-lite-lambda
```

```
docker tag tf-lite-lambda ${REMOTE_NAME}
```

```
docker push ${REMOTE_NAME}
```

Creating the lambda function



- Next, click Create Function. Select Container Image

Create function Info

Choose one of the following options to create your function.

Author from scratch
Start with a simple Hello World example.

Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.

Container image
Select a container image to deploy for your function.

Browse serverless app repository
Deploy a sample Lambda application from the AWS Serverless Application Repository.

Creating the lambda function

- After that, fill in the details

Basic information

Function name
Enter a name that describes the purpose of your function.

`clothes-classification`

Use only letters, numbers, hyphens, or underscores with no spaces.

Container image URI Info
The location of the container image to use for your function.

`<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/lambda-images:tf-lite-lambda`

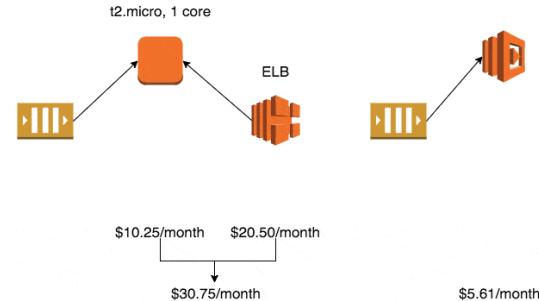
Requires a valid Amazon ECR Image URI.

[Browse images](#)

Creating the lambda function

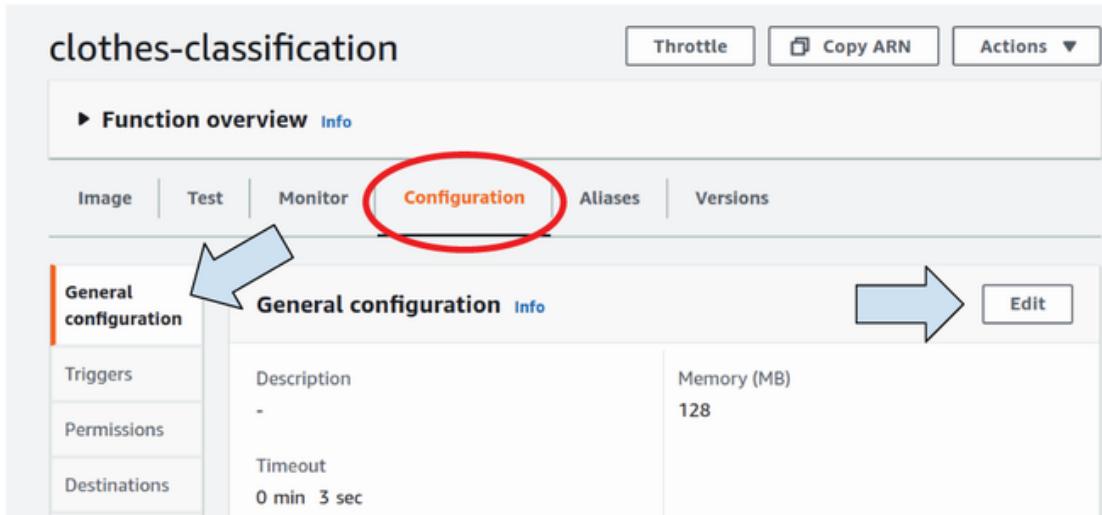
- The container image URI should be the image we created earlier and pushed to ECR:

<ACCOUNT>.dkr.ecr.<REGION>.amazonaws.com/lambda-images:tf-lite-lambda



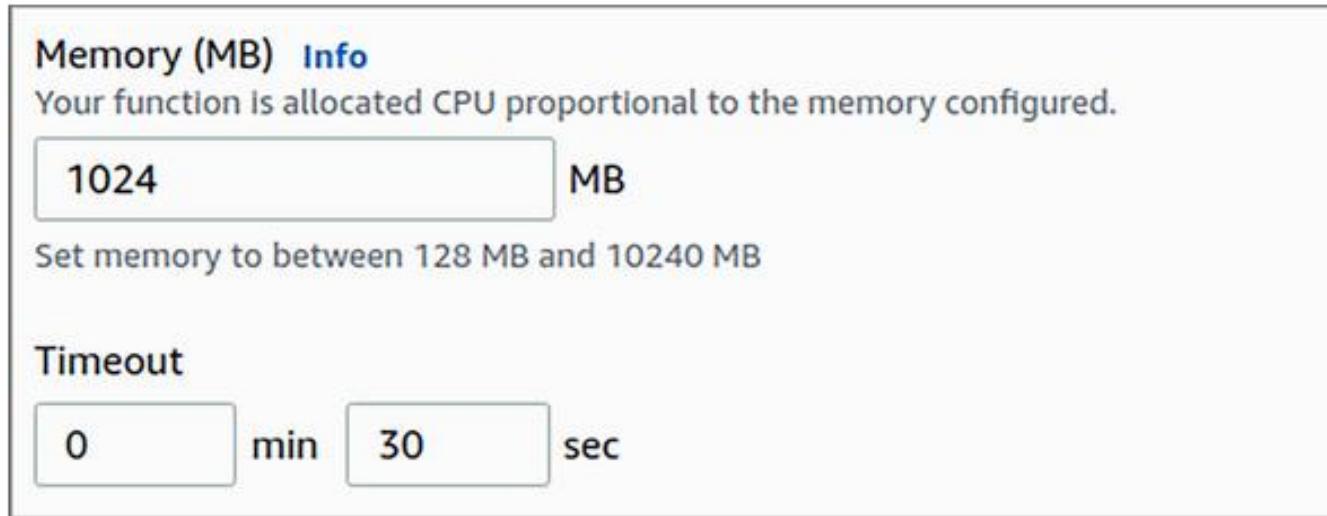
Creating the lambda function

- Select the Configuration tab, choose General Configuration, and then click Edit.



Creating the lambda function

- Click the Edit button, give it 1024 MB of RAM, and set the timeout to 30 seconds.



Creating the lambda function

Test tab

The screenshot shows the 'Test' tab of the AWS Lambda function configuration interface. The top navigation bar includes tabs for 'Image', 'Test' (which is highlighted with a red oval), 'Monitor', 'Configuration', 'Aliases', and 'Versions'. Below the tabs, there's a section for defining a 'Test event'. It features three buttons: 'Format', 'Save changes', and a prominent orange 'Test' button. A text input field below these buttons contains the instruction: 'Invoke your function with a test event. Choose a template that matches the service that triggers your function, or enter your event document in JSON.' Underneath this, there are two radio button options: 'New event' (selected) and 'Saved event'. A dropdown menu labeled 'Template' is set to 'hello-world'. The 'Name' field contains the value 'test'. A blue arrow points from the 'Name' field towards the JSON code editor. The JSON code editor itself displays the following document:

```
1 {  
2   "url": "http://bit.ly/mlbookcamp-pants"  
3 }
```

A blue arrow points from the right side of the JSON code editor towards the bottom right corner of the interface.

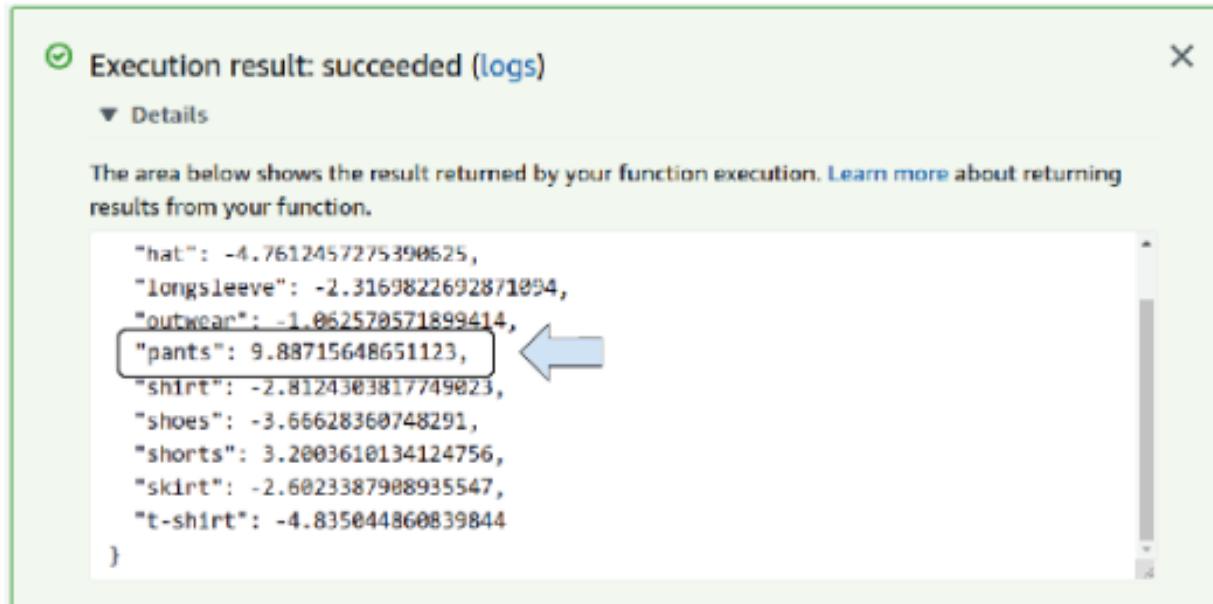
Creating the lambda function

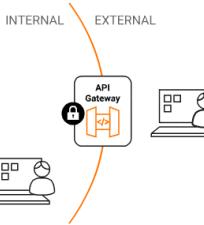
- Create a test event. Give it a name (for example, test), and put the following content in the request body:

```
{  
  "url": "http://bit.ly/mlbookcamp-pants"  
}
```

Creating the lambda function

- Execution results: succeeded





Creating the API Gateway

- In the AWS Console, find the API Gateway service. Create a new API: select REST API, and click Build.
- Then select New API, and call it clothes-classification. Click Create API.

Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

New API Import from Swagger or Open API 3 Example API

Settings

Choose a friendly name and description for your API.

API name*	clothes-classification
Description	expose lambda as a web service
Endpoint Type	Regional

Creating the API Gateway

- Next, click the Actions button and select Resource. Then, create a resource predict.

Configure as [proxy resource](#) ⓘ

Resource Name*

Resource Path*

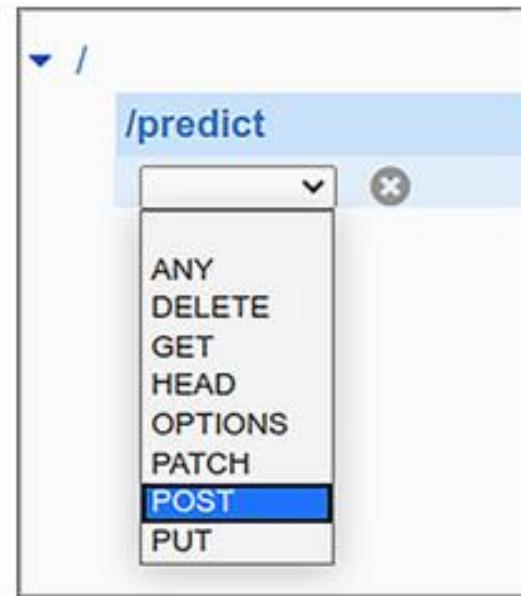
You can add path parameters using brackets. For example, the resource path `{username}` represents a path parameter called 'username'. Configuring `/{proxy+}` as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to `/foo`. To handle requests to `/`, add a new ANY method on the `/` resource.

Enable API Gateway CORS ⓘ

Creating the API Gateway

After creating the resource, create a POST method for it:

- Click Predict.
- Click Actions.
- Select Create Method.
- Choose POST from the list.
- Click the tick button.



Creating the API Gateway

- Now select Lambda Function as the integration type and enter the name of your lambda function.

The screenshot shows the 'Lambda Function' configuration section of the AWS API Gateway. At the top, there's a radio button group for 'Integration type' with 'Lambda Function' selected (indicated by a blue dot). Below it are five other options: 'HTTP', 'Mock', 'AWS Service', and 'VPC Link', each with a small info icon. A note below says 'Use Lambda Proxy integration' with an unchecked checkbox and an info icon. A curved arrow points from this note to the text 'Make sure it's NOT checked.' In the 'Lambda Region' dropdown, 'eu-west-1' is selected. The 'Lambda Function' input field contains the text 'clothes-classification'. Below the input field is a checked checkbox for 'Use Default Timeout' with an info icon. At the bottom right is a blue 'Save' button.

Integration type Lambda Function ⓘ

HTTP ⓘ

Mock ⓘ

AWS Service ⓘ

VPC Link ⓘ

Use Lambda Proxy integration ⓘ ← Make sure it's **NOT** checked.

Lambda Region eu-west-1

Lambda Function

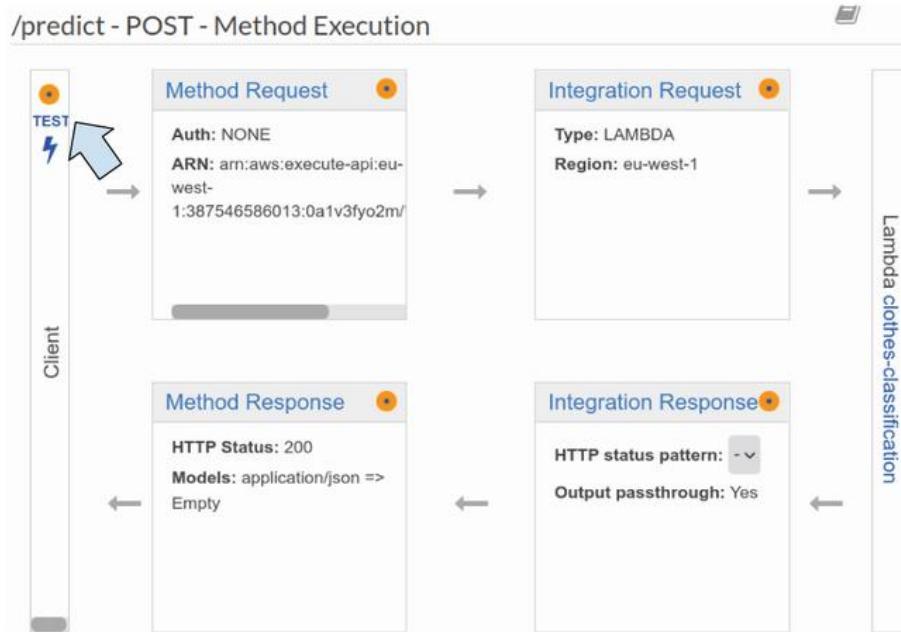
clothes-classification ⓘ

Use Default Timeout ⓘ

Save

Creating the API Gateway

- After doing this, we should see the integration.



Creating the API Gateway

- Click TEST, and put the same request in the request body as previously:

```
{  
  "url": "http://bit.ly/mlbookcamp-pants"  
}
```

- The response is the same: the predicted class is *pants*.

Request: /predict

Status: 200

Latency: 5327 ms

Response Body

```
{  
  "dress": -1.8682900667190552,  
  "hat": -4.7612457275390625,  
  "longsleeve": -2.3169822692871094,  
  "outwear": -1.062570571899414,  
  "pants": 9.88715648651123, pants ←  
  "shirt": -2.8124303817749023,  
  "shoes": -3.66628360748291,  
  "shorts": 3.2003610134124756,  
  "skirt": -2.6023387908935547,  
  "t-shirt": -4.835044860839844  
}
```

Creating the API Gateway

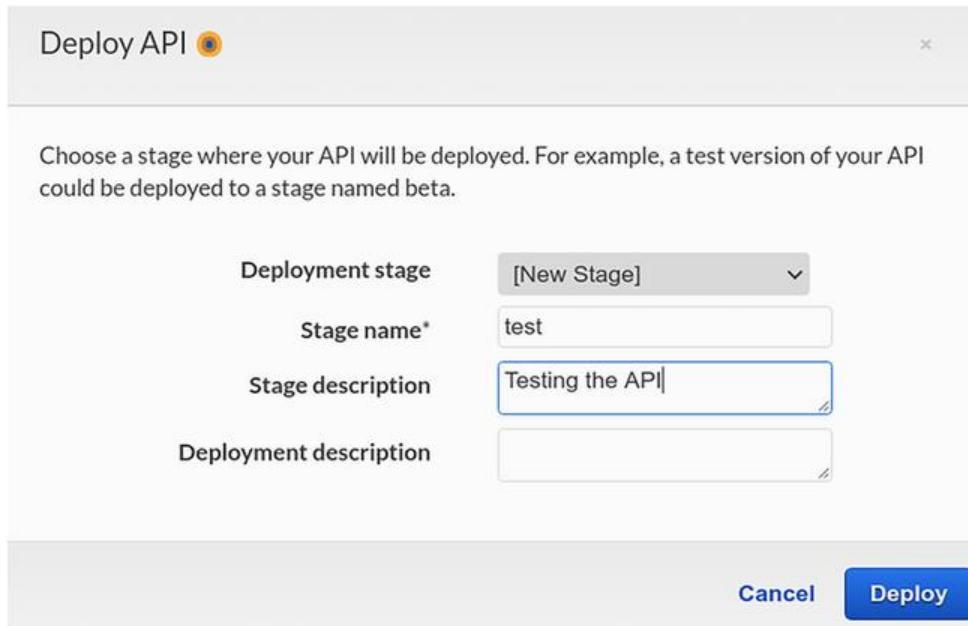
- To use it externally, we need to deploy the API. Select Deploy API from the list of actions.

The screenshot shows the AWS API Gateway Actions menu for a POST method named /predict. The 'Actions' dropdown is open, revealing three sections: METHOD ACTIONS, RESOURCE ACTIONS, and API ACTIONS. In the METHOD ACTIONS section, 'Delete Method' is highlighted in red. In the RESOURCE ACTIONS section, 'Delete Resource' is highlighted in red. In the API ACTIONS section, 'Deploy API' is highlighted in red and has a blue arrow pointing to it. The URL in the browser bar is /predict - POST.

Actions	
Edit Method Documentation	
Delete Method	
Create Method	
Create Resource	
Enable CORS	
Edit Resource Documentation	
Delete Resource	
Deploy API	
Import API	

Creating the API Gateway

- Next, create a new stage test.



Creating the API Gateway

- Let's take the test.py script we created previously and replace the URL there:

```
import requests
```

```
data = {  
    "url": "http://bit.ly/mlbookcamp-pants"  
}
```

```
url = "https://0a1v3fy02m.execute-api.eu-west-1.amazonaws.com/test/predict"
```

```
results = requests.post(url, json=data).json()
```

```
print(results)
```

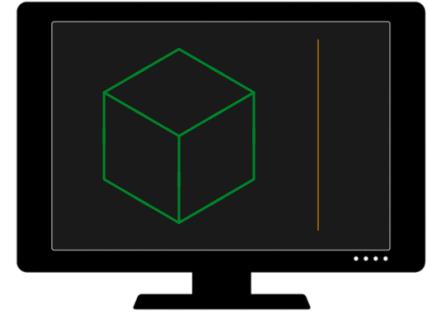
Creating the API Gateway

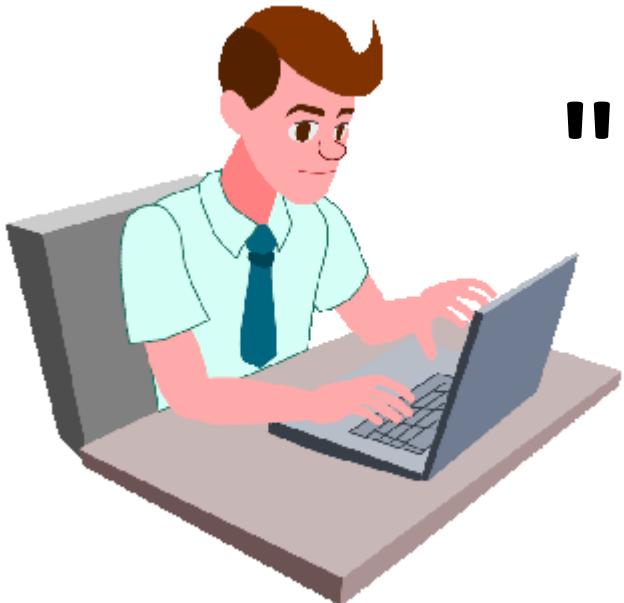
- Run it:

```
python test.py
```

- The response is the same as previously:

```
{  
    "dress": -1.86829,  
    "hat": -4.76124,  
    "longsleeve": -2.31698,  
    "outwear": -1.06257,  
    "pants": 9.88715,  
    "shirt": -2.81243,  
    "shoes": -3.66628,  
    "shorts": 3.20036,  
    "skirt": -2.60233,  
    "t-shirt": -4.83504  
}
```





"Complete Exercise"

Summary



- TensorFlow Lite is a lightweight alternative to “full” TensorFlow. It contains only the most important parts that are needed for using deep learning models. Using it makes the process of deploying models with AWS Lambda faster and simpler.
- Lambda functions can be run locally using Docker. This way, we can test our code without deploying it to AWS.
- To deploy a lambda function, we need to put its code in Docker, publish the Docker image to ECR, and then use the URI of the image when creating a lambda function.
- To expose the lambda function, we use API Gateway. This way, we make the lambda function available as a web service, so it could be used by anyone.