Learning ML Model with the Titanic.

Overview

A detailed ML workflow with Scikit Learn using the data from the Titanic maritime tragedy.

Introduction

The focus of this experiment is you can use Vertex AI to train and deploy a ML model. It assumes that you are familiar with Machine Learning even though the machine learning code for training is provided to you. You will use Datasets for dataset creation and management, and custom model for training a Scikit Learn model. Finally, you will deploy the trained model and get online predictions. The dataset you will use for this demo is the Titanic Dataset.

Setup your environment

Project

To complete this experiment you need a Google Cloud Platform project. We use the same project for all our experiments, but in production you'd likely use a separate project for each major initiative.

Load data in BigQuery

In order to train a Machine Learning model you need access to data. BigQuery is a serverless, highly scalable, and cost-effective multi-cloud data warehouse and it is the perfect service for keeping your data.

Create dataset

To create a BigQuery dataset, navigate to BigQuery on Google Cloud Console

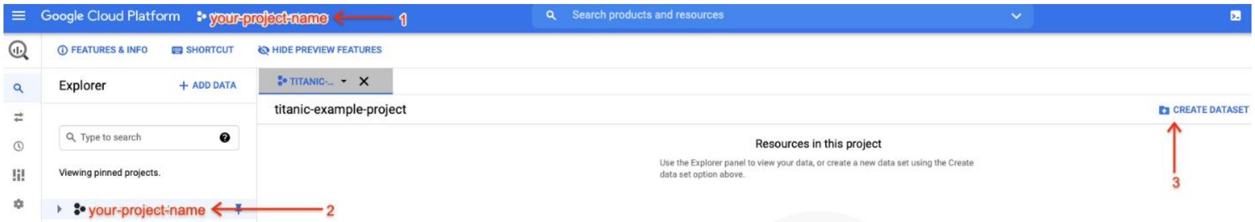
The below image marks 3 steps:

1. Make sure that you select the right project from the top of console page
2. Select the project you want to create the Dataset in

3. Click **Create Dataset**

A popup will appear. Enter the *dataset id*: **titanic** and then click **Create Dataset**

You have now created the dataset.



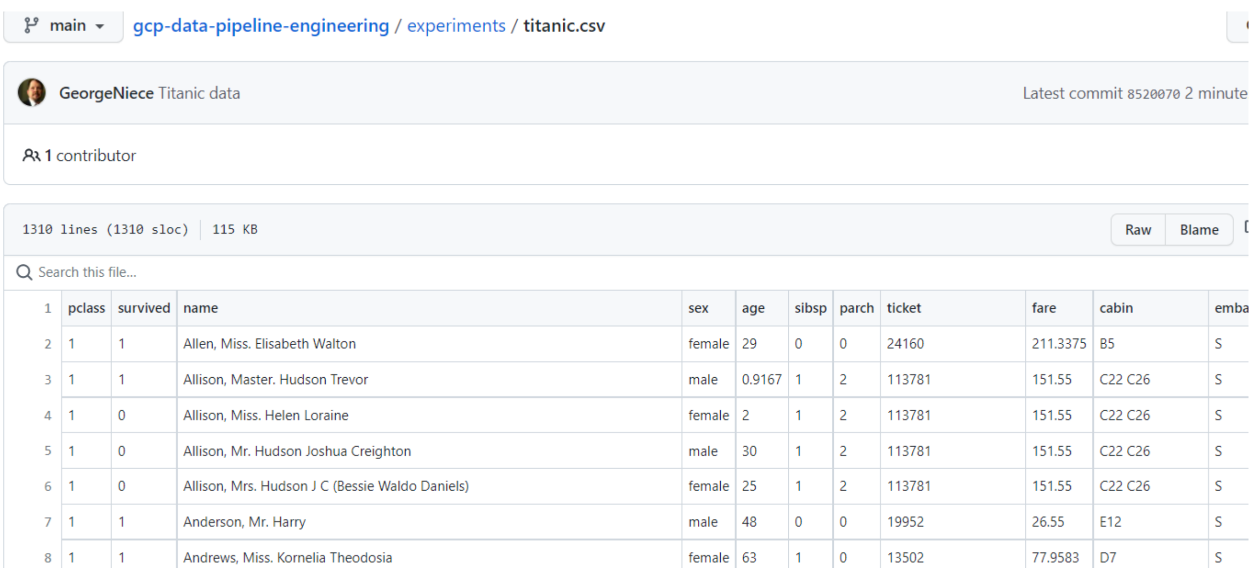
Create table

You need a table to load your data. First download the Titanic dataset locally.

curl -O https://raw.githubusercontent.com/GeorgeNiece/gcp-data-pipeline-engineering/main/experiments/titanic.csv

Alternatively you can just download the file from our session GitHub repository at

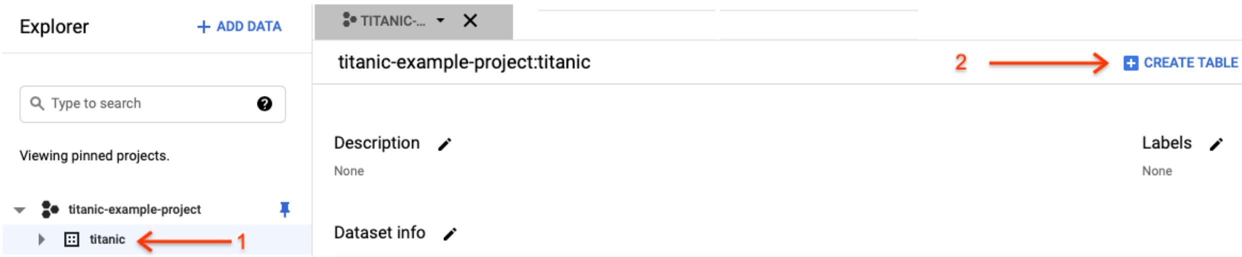
https://github.com/GeorgeNiece/gcp-data-pipeline-engineering/blob/main/experiments/titanic.csv



	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	emba
1	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	emba
2	1	1	Allen, Miss. Elisabeth Walton	female	29	0	0	24160	211.3375	B5	S
3	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.55	C22 C26	S
4	1	0	Allison, Miss. Helen Loraine	female	2	1	2	113781	151.55	C22 C26	S
5	1	0	Allison, Mr. Hudson Joshua Creighton	male	30	1	2	113781	151.55	C22 C26	S
6	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25	1	2	113781	151.55	C22 C26	S
7	1	1	Anderson, Mr. Harry	male	48	0	0	19952	26.55	E12	S
8	1	1	Andrews, Miss. Kornelia Theodosia	female	63	1	0	13502	77.9583	D7	S

Then from the UI :

1. Select the **titanic** dataset you have created in the previous step
2. Click **CREATE TABLE**



From the Sidebar select the following:

1. Create table from: **Upload**
2. Select file: *Use the downloaded titanic dataset*
3. File Format: **CSV**
4. Table Name: **survivors**
5. Auto-detect: Select auto-detect checkbox - **Schema and input parameters**

Click **Create Table**

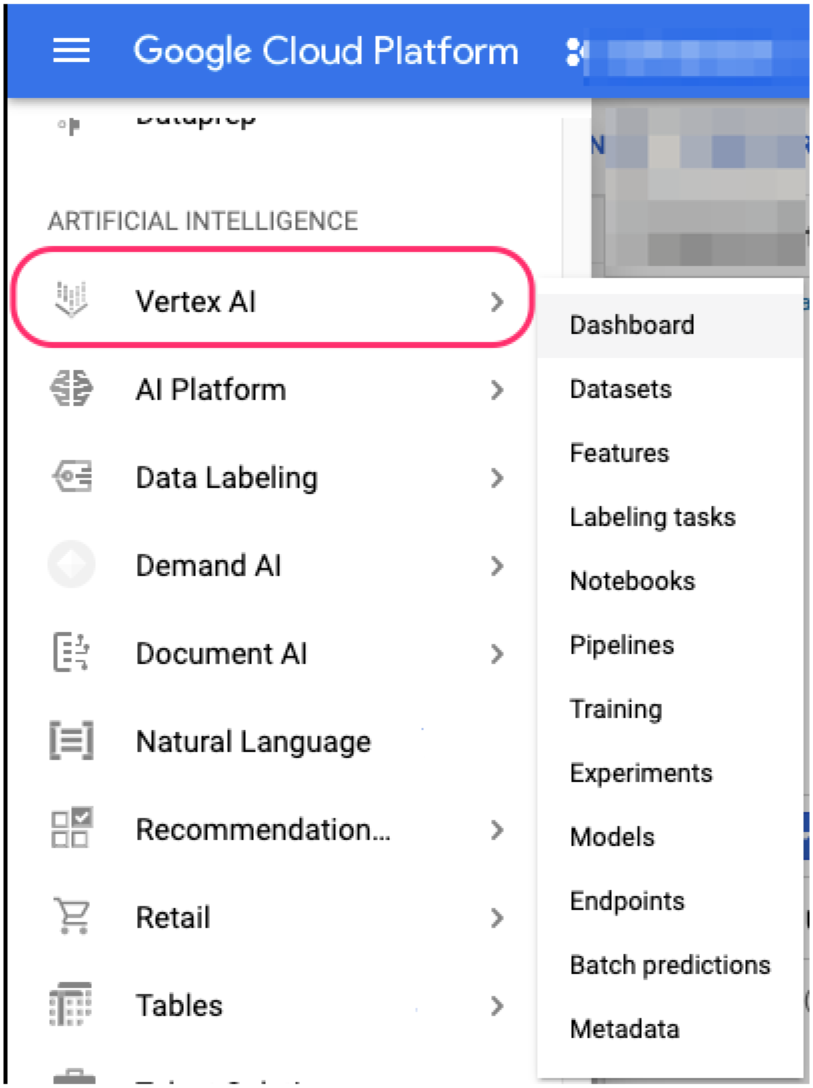
You have now created and populated the table with the titanic dataset! You can explore the table contents, run queries and analyse your data.

Create a dataset

Datasets in Vertex AI allow you to create datasets for your Machine Learning workloads. You can create datasets for structured data (CSV files or BigQuery tables) or unstructured data such as Images and Text. It is important to notice that Vertex AI Datasets just reference your original data and there is no duplication.

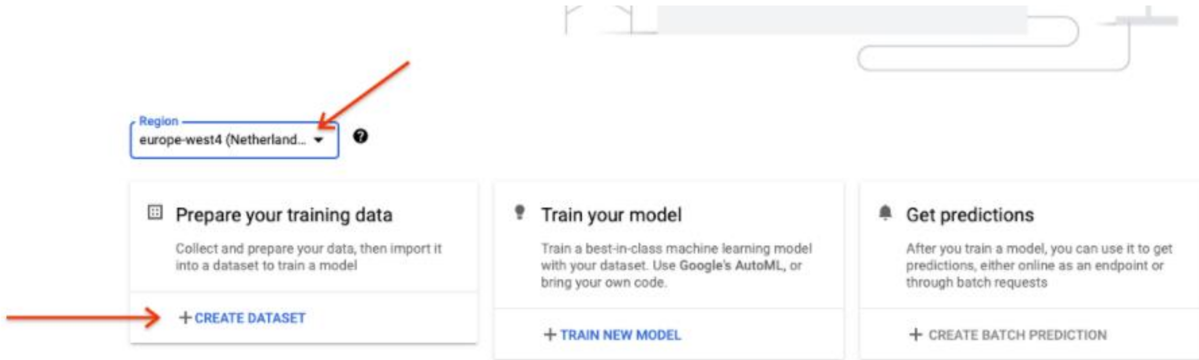
Create ML dataset

Find Vertex AI on the GCP side menu, under Artificial Intelligence. If this is the first time visiting Vertex AI, you will get a notification to Enable Vertex AI API. Please do so!



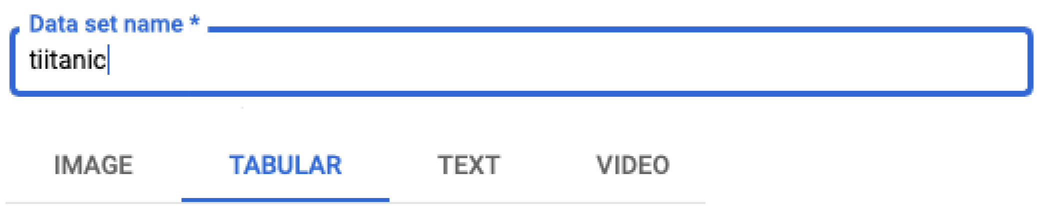
Once you select Vertex AI you can select a region you want your resources to use. Thus tutorial is using europe-west4 as a reagion. If you need to use a different regions you can. Just replace europe-west4 with the region of your choice for the rest of this tutorial.

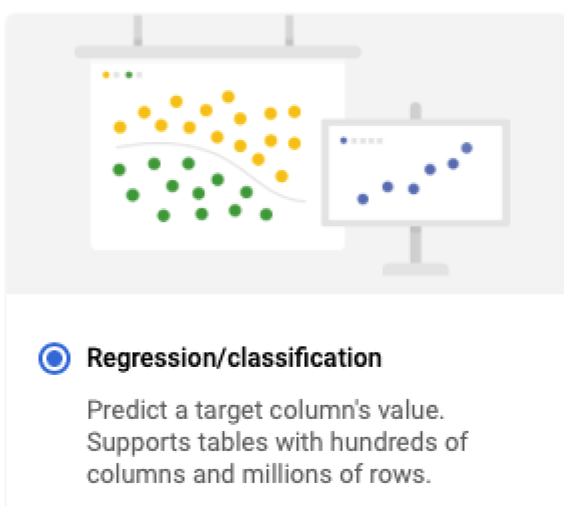
Select **europe-west4** and click on **CREATE A DATASET**



Give your dataset a name. How about **titanic**

You can create datasets for images, text or videos as well as tabular data. The titanic dataset is tabular so you should click the Tabular tab

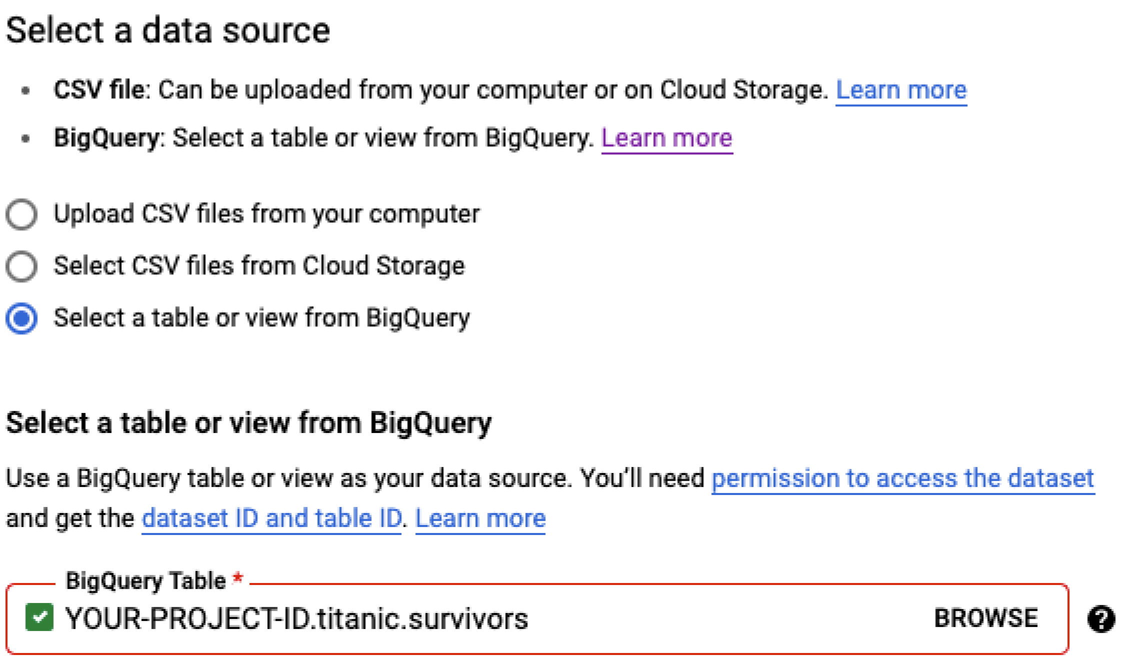




For region selection select **europe-west4** and click **CREATE** . We did not yet connect to the datasource yet. We just created a placeholder. Your will connect the datasource on the following step.

Select datasource

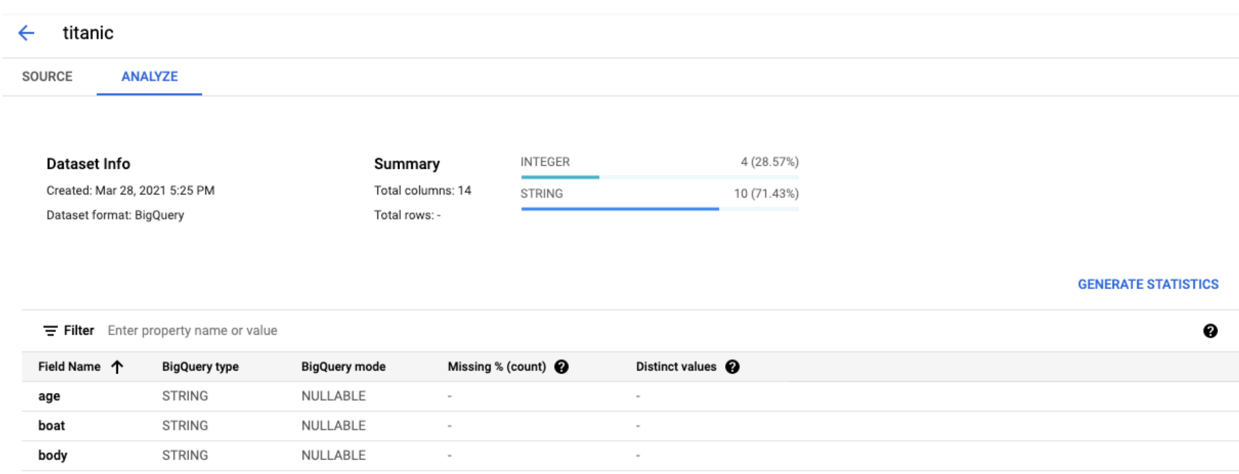
As you have already loaded the titanic dataset in BigQuery, we can connect our ML dataset to our BigQuery table as shown in the image. To make it easy to find your table you can click **Browse**. Once you select the dataset click on **CONTINUE**



Generate statistics

Under the **ANALYZE** tab you can generate statistics regarding your data. This gives you the ability to quickly have a peek at the data and check for distributions, missing values etc.

In order to run the statistical analysis click **GENERATE STATISTICS**. It can take a couple of minutes to execute. You can continue with the lab and came back later to see the results.



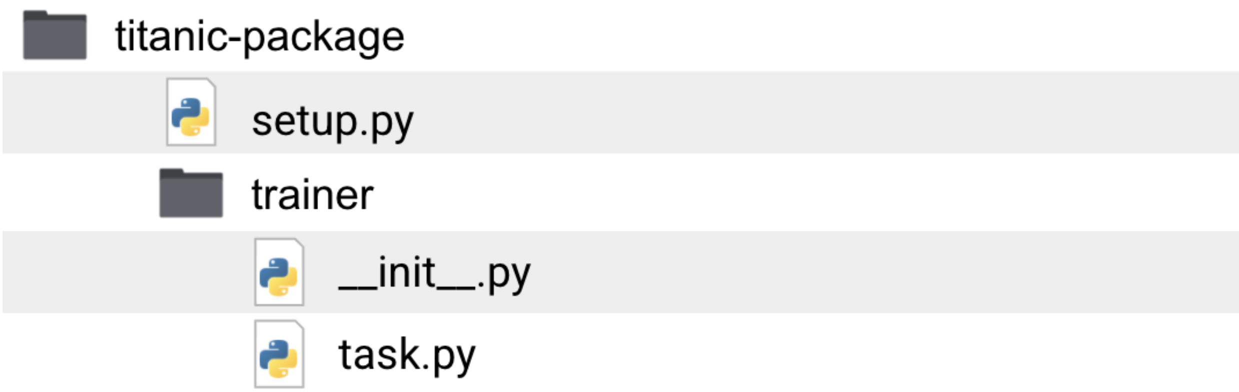
Custom training package using Notebooks

It is a good practice to package and parameterise your code so that it becomes a portable asset.

In this section you will create a training package with custom code using Notebooks. A fundamental step using the service is to be able to create a python source distribution, AKA a distribution package. This is not much more than creating folders and files within. The next section will explain how a package is structured.

Application Structure

The basic structured of a python package can be seen in the image below.



Let's see what those folders and files are for:

- **titanic-package**: This is your working directory. Inside this folder we will have our package and code related to the titanic survivor classifier.
- **setup.py**: The setup file specifies how to build your distribution package. It includes information such as the package name, version as well as any other packages that you might need for your training job and are not included by default in GCP's pre-built training containers.
- **trainer**: The folder that contains the training code. This is also a python package. What makes it a package is the empty __init__.py file that is inside the folder.
- **__init__.py**: Empty file called __init__.py. It signifies that the folder that it belongs to is a package.
- **task.py**: The task.py is a package module. Here is the entry point of your code and it also accepts CLI parameters for model training. You can include your

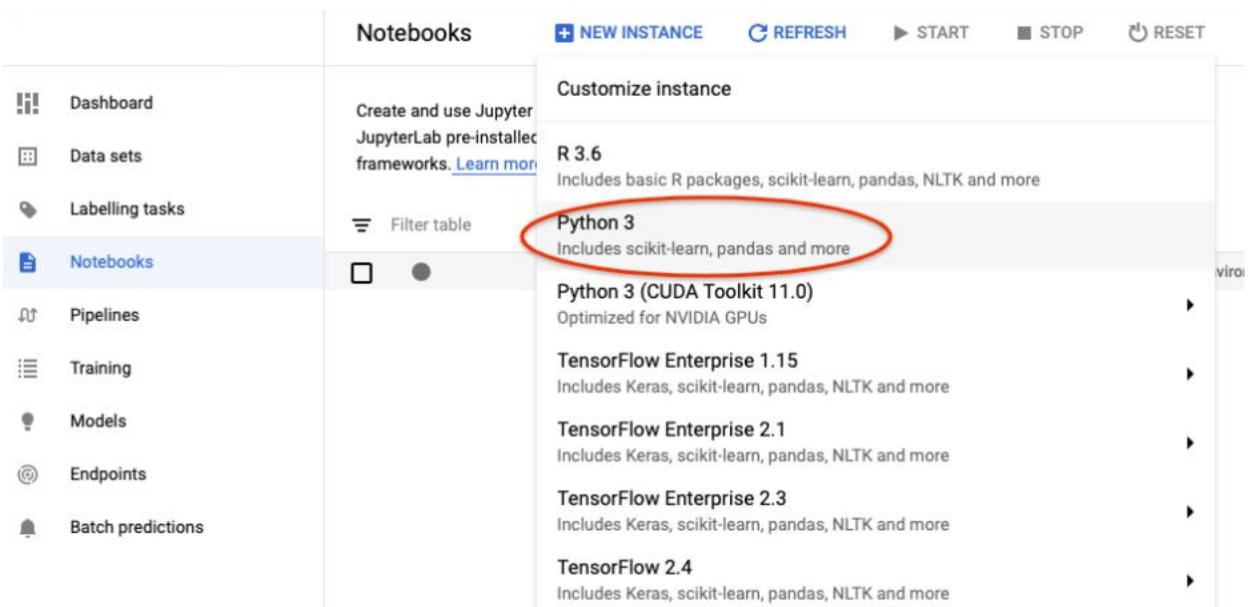
training code in this module as well or you can create additional modules inside your package. This is entirely up to you and how you want to structure your code.

Now that you have an understanding of the structure, let's clarify that the names used for the package and module do not have to be trainer and task.py. We only use this naming convention so that it aligns with our experiment but you can in fact pick the names that suit you.

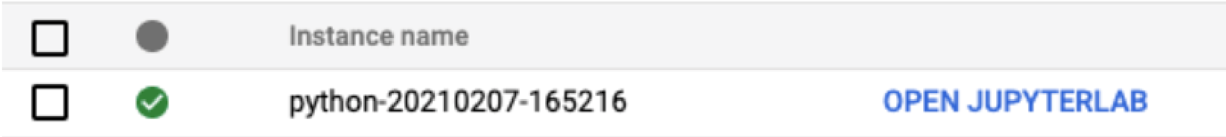
Create your notebook instance

How about creating a notebook instance and try training a custom model? From the Vertex AI navigate to notebooks and start an instance with **Python 3**, which includes scikit-learn as shown in the image below. We will use a scikit learn model for our classifier.

A pop-up will appear. Here you can change settings like the region your notebook instance will be created at and the compute power you require. As we are not dealing with a lot of data and the we only need the instance for development purposes please do not change any of the settings and simply click **Create**



The instance will be up and running in no more than a couple of minutes. Once the instance is ready go ahead and **OPEN JUPYTERLAB**



Create your package

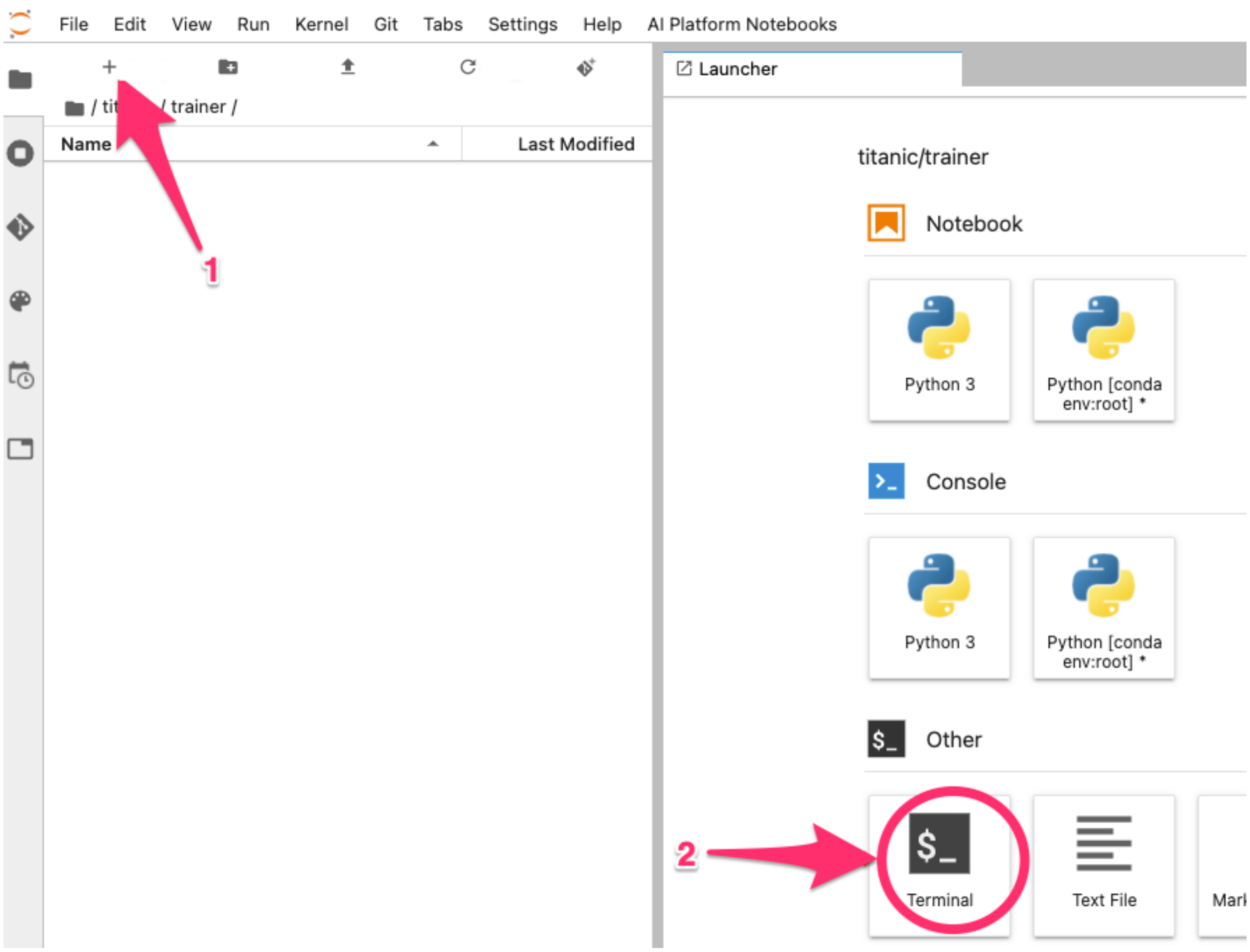
Now that the notebook is up and running you can start building your training assets.

For this task is easier to use the terminal. From the Launcher, click on Terminal to create a new terminal session (marks #1 and #2 in the image below)

Now in the terminal execute the following commands to create the folder structure with the required files

```
mkdir -p /home/jupyter/titanic/trainer
touch /home/jupyter/titanic/setup.py
/home/jupyter/titanic/trainer/__init__.py
/home/jupyter/titanic/trainer/task.py
```

Once you run the commands click the refresh button (#3 in below image) to see the newly created folder and files



Copy-paste the following code in titanic/trainer/task.py The code contains comments. Spend few minutes going through the file to better understand it:

```
from google.cloud import bigquery, bigquery_storage, storage
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder,
OrdinalEncoder
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC
from sklearn.metrics import classification_report, f1_score
from typing import Union, List
import os, logging, json, pickle, argparse
import dask.dataframe as dd
import pandas as pd
import numpy as np

# feature selection.  The FEATURE list defines what features are
needed from the training data.
# as well as the types of those features. We will perform different
```

```
feature engineering depending on the type

# List all column names for binary features: 0,1 or True,False or
Male,Female etc
BINARY_FEATURES = [
    'sex']

# List all column names for numeric features
NUMERIC_FEATURES = [
    'age',
    'fare']

# List all column names for categorical features
CATEGORICAL_FEATURES = [
    'pclass',
    'embarked',
    'home_dest',
    'parch',
    'sibsp']

ALL_COLUMNS = BINARY_FEATURES+NUMERIC_FEATURES+CATEGORICAL_FEATURES

# define the column name for label
LABEL = 'survived'

# Define the index position of each feature. This is needed for
processing a
# numpy array (instead of pandas) which has no column names.
BINARY_FEATURES_IDX = list(range(0,len(BINARY_FEATURES)))
NUMERIC_FEATURES_IDX = list(range(len(BINARY_FEATURES),
len(BINARY_FEATURES)+len(NUMERIC_FEATURES)))
CATEGORICAL_FEATURES_IDX =
list(range(len(BINARY_FEATURES+NUMERIC_FEATURES), len(ALL_COLUMNS)))


def load_data_from_gcs(data_gcs_path: str) -> pd.DataFrame:
    '''
    Loads data from Google Cloud Storage (GCS) to a dataframe

            Parameters:
                    data_gcs_path (str): gs path for the location of
the data. Wildcards are also supported. i.e
gs://example_bucket/data/training-*.csv

            Returns:
                    pandas.DataFrame: a dataframe with the data from
GCP loaded
    '''
```

```
    # using dask that supports wildcards to read multiple files. Then
with dd.read_csv().compute we create a pandas dataframe
    # Additionally I have noticed that some values for TotalCharges
are missing and this creates confusion regarding TotalCharges the data
types.
    # to overcome this we manually define TotalCharges as object.
    # We will later fix this upnormality
    logging.info("reading gs data: {}".format(data_gcs_path))
    return dd.read_csv(data_gcs_path, dtype={'TotalCharges':
'object'}).compute()


def load_data_from_bq(bq_uri: str) -> pd.DataFrame:
    '''
    Loads data from BigQuery table (BQ) to a dataframe

            Parameters:
                    bq_uri (str): bq table uri. i.e:
example_project.example_dataset.example_table
            Returns:
                    pandas.DataFrame: a dataframe with the data from
GCP loaded
    '''
    if not bq_uri.startswith('bq://'):
        raise Exception("uri is not a BQ uri. It should be
bq://project_id.dataset.table")
    logging.info("reading bq data: {}".format(bq_uri))
    project,dataset,table =  bq_uri.split(".")
    bqclient = bigquery.Client(project=project[5:])
    bqstorageclient = bigquery_storage.BigQueryReadClient()
    query_string = """
    SELECT * from {ds}.{tbl}
    """.format(ds=dataset, tbl=table)

    return (
        bqclient.query(query_string)
        .result()
        .to_dataframe(bqstorage_client=bqstorageclient)
    )

def clean_missing_numerics(df: pd.DataFrame, numeric_columns):
    '''
    removes invalid values in the numeric columns

            Parameters:
                    df (pandas.DataFrame): The Pandas Dataframe to
alter
```

```
                    numeric_columns (List[str]): List of column names
that are numberic from the DataFrame
            Returns:
                    pandas.DataFrame: a dataframe with the numeric
columns fixed
    '''

    for n in numeric_columns:
        df[n] = pd.to_numeric(df[n], errors='coerce')

    df = df.fillna(df.mean())

    return df

def data_selection(df: pd.DataFrame, selected_columns: List[str],
label_column: str) -> (pd.DataFrame, pd.Series):
    '''
    From a dataframe it creates a new dataframe with only selected
columns and returns it.
    Additionally it splits the label column into a pandas Series.

            Parameters:
                    df (pandas.DataFrame): The Pandas Dataframe to
drop columns and extract label
                    selected_columns (List[str]): List of strings with
the selected columns. i,e ['col_1', 'col_2', ..., 'col_n' ]
                    label_column (str): The name of the label column

            Returns:
                    tuple(pandas.DataFrame, pandas.Series): Tuble with
the new pandas DataFrame containing only selected columns and lablel
pandas Series
    '''
    # We create a series with the prediciton label
    labels = df[label_column]

    data = df.loc[:, selected_columns]


    return data, labels

def pipeline_builder(params_svm: dict, bin_ftr_idx: List[int],
num_ftr_idx: List[int], cat_ftr_idx: List[int]) -> Pipeline:
    '''
    Builds a sklearn pipeline with preprocessing and model
configuration.
    Preprocessing steps are:
        * OrdinalEncoder - used for binary features
```

```
        * StandardScaler - used for numerical features
        * OneHotEncoder - used for categorical features
    Model used is SVC

            Parameters:
                    params_svm (dict): List of parameters for the
sklearn.svm.SVC classifier
                    bin_ftr_idx (List[str]): List of ints that mark
the column indexes with binary columns. i.e [0, 2, ... , X ]
                    num_ftr_idx (List[str]): List of ints that mark
the column indexes with numerica columns. i.e [6, 3, ... , X ]
                    cat_ftr_idx (List[str]): List of ints that mark
the column indexes with categorical columns. i.e [5, 10, ... , X ]
                    label_column (str): The name of the label column

            Returns:
                     Pipeline: sklearn.pipelines.Pipeline with
preprocessing and model training
    '''

    # Definining a preprocessing step for our pipeline.
    # it specifies how the features are going to be transformed
    preprocessor = ColumnTransformer(
        transformers=[
            ('bin', OrdinalEncoder(), bin_ftr_idx),
            ('num', StandardScaler(), num_ftr_idx),
            ('cat', OneHotEncoder(handle_unknown='ignore'),
cat_ftr_idx)], n_jobs=-1)


    # We now create a full pipeline, for preprocessing and training.
    # for training we selected a linear SVM classifier

    clf = SVC()
    clf.set_params(**params_svm)

    return Pipeline(steps=[ ('preprocessor', preprocessor),
                          ('classifier', clf)])

def train_pipeline(clf: Pipeline, X: Union[pd.DataFrame, np.ndarray],
y: Union[pd.DataFrame, np.ndarray]) -> float:
    '''
    Trains a sklearn pipeline by fiting training data an labels and
returns the accuracy f1 score

            Parameters:
                    clf (sklearn.pipelines.Pipeline): the Pipeline
object to fit the data
```

```
                    X: (pd.DataFrame OR np.ndarray): Training vectors
of shape n_samples x n_features, where n_samples is the number of
samples and n_features is the number of features.
                    y: (pd.DataFrame OR np.ndarray): Labels of shape
n_samples. Order should mathc Training Vectors X

            Returns:
                    score (float): Average F1 score from all cross
validations
    '''
    # run cross validation to get training score. we can use this
score to optimise training
    score = cross_val_score(clf, X, y, cv=10, n_jobs=-1).mean()

    # Now we fit all our data to the classifier.
    clf.fit(X, y)

    return score

def process_gcs_uri(uri: str) -> (str, str, str, str):
    '''
    Receives a Google Cloud Storage (GCS) uri and breaks it down to
the scheme, bucket, path and file

            Parameters:
                    uri (str): GCS uri

            Returns:
                    scheme (str): uri scheme
                    bucket (str): uri bucket
                    path (str): uri path
                    file (str): uri file
    '''
    url_arr = uri.split("/")
    if "." not in url_arr[-1]:
        file = ""
    else:
        file = url_arr.pop()
    scheme = url_arr[0]
    bucket = url_arr[2]
    path = "/".join(url_arr[3:])
    path = path[:-1] if path.endswith("/") else path

    return scheme, bucket, path, file

def pipeline_export_gcs(fitted_pipeline: Pipeline, model_dir: str) ->
str:
    '''
```

```
    Exports trained pipeline to GCS

            Parameters:
                    fitted_pipeline (sklearn.pipelines.Pipeline): the
Pipeline object with data already fitted (trained pipeline object)
                    model_dir (str): GCS path to store the trained
pipeline. i.e gs://example_bucket/training-job
            Returns:
                    export_path (str): Model GCS location
    '''
    scheme, bucket, path, file = process_gcs_uri(model_dir)
    if scheme != "gs:":
            raise ValueError("URI scheme must be gs")

    # Upload the model to GCS
    b = storage.Client().bucket(bucket)
    export_path = os.path.join(path, 'model.pkl')
    blob = b.blob(export_path)

    blob.upload_from_string(pickle.dumps(fitted_pipeline))
    return scheme + "//" + os.path.join(bucket, export_path)


def prepare_report(cv_score: float, model_params: dict,
classification_report: str, columns: List[str], example_data:
np.ndarray) -> str:
    '''
    Prepares a training report in Text

            Parameters:
                    cv_score (float): score of the training job during
cross validation of training data
                    model_params (dict): dictonary containing the
parameters the model was trained with
                    classification_report (str): Model classification
report with test data
                    columns (List[str]): List of columns that where
used in training.
                    example_data (np.array): Sample of data (2-3 rows
are enough). This is used to include what the prediciton payload
should look like for the model
            Returns:
                    report (str): Full report in text
    '''

    buffer_example_data = '['
    for r in example_data:
        buffer_example_data+='['
```

```
        for c in r:
            if(isinstance(c,str)):
                buffer_example_data+="'"+c+"', "
            else:
                buffer_example_data+=str(c)+", "
        buffer_example_data= buffer_example_data[:-2]+"], \n"
    buffer_example_data= buffer_example_data[:-3]+"]"

    report = """
Training Job Report

Cross Validation Score: {cv_score}

Training Model Parameters: {model_params}

Test Data Classification Report:
{classification_report}

Example of data array for prediciton:

Order of columns:
{columns}

Example for clf.predict()
{predict_example}


Example of GCP API request body:
{{
    "instances": {json_example}
}}

""".format(
    cv_score=cv_score,
    model_params=json.dumps(model_params),
    classification_report=classification_report,
    columns = columns,
    predict_example = buffer_example_data,
    json_example = json.dumps(example_data.tolist()))

    return report


def report_export_gcs(report: str, report_dir: str) -> None:
    '''
    Exports training job report to GCS

            Parameters:
```

```
                    report (str): Full report in text to sent to GCS
                    report_dir (str): GCS path to store the report
model. i.e gs://example_bucket/training-job
            Returns:
                    export_path (str): Report GCS location
    '''
    scheme, bucket, path, file = process_gcs_uri(report_dir)
    if scheme != "gs:":
            raise ValueError("URI scheme must be gs")

    # Upload the model to GCS
    b = storage.Client().bucket(bucket)

    export_path = os.path.join(path, 'report.txt')
    blob = b.blob(export_path)

    blob.upload_from_string(report)

    return scheme + "//" + os.path.join(bucket, export_path)



# Define all the command line arguments your model can accept for
training
if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    # Input Arguments

    parser.add_argument(
        '--model_param_kernel',
        help = 'SVC model parameter- kernel',
        choices=['linear', 'poly', 'rbf', 'sigmoid', 'precomputed'],
        type = str,
        default = 'linear'
    )

    parser.add_argument(
        '--model_param_degree',
        help = 'SVC model parameter- Degree. Only applies for poly
kernel',
        type = int,
        default = 3
    )

    parser.add_argument(
        '--model_param_C',
        help = 'SVC model parameter- C (regularization)',
```

```
        type = float,
        default = 1.0
    )

    parser.add_argument(
        '--model_param_probability',
        help = 'Whether to enable probability estimates',
        type = bool,
        default = True
    )


    '''
    Vertex AI automatically populates a set of environment varialbes
in the container that executes
    your training job. those variables include:
        * AIP_MODEL_DIR - Directory selected as model dir
        * AIP_DATA_FORMAT - Type of dataset selected for training (can
be csv or bigquery)

    Vertex AI will automatically split selected dataset into
training,validation and testing
    and 3 more environment variables will reflect the locaiton of the
data:
        * AIP_TRAINING_DATA_URI - URI of Training data
        * AIP_VALIDATION_DATA_URI - URI of Validation data
        * AIP_TEST_DATA_URI - URI of Test data

    Notice that those environment varialbes are default. If the user
provides a value using CLI argument,
    the environment variable will be ignored. If the user does not
provide anything as CLI  argument
    the program will try and use the environemnt variables if those
exist. otherwise will leave empty.
    '''
    parser.add_argument(
        '--model_dir',
        help = 'Directory to output model and artifacts',
        type = str,
        default = os.environ['AIP_MODEL_DIR'] if 'AIP_MODEL_DIR' in
os.environ else ""
    )
    parser.add_argument(
        '--data_format',
        choices=['csv', 'bigquery'],
        help = 'format of data uri csv for gs:// paths and bigquery
for project.dataset.table formats',
        type = str,
```

```
        default =  os.environ['AIP_DATA_FORMAT'] if 'AIP_DATA_FORMAT'
in os.environ else "csv"
    )
    parser.add_argument(
        '--training_data_uri',
        help = 'location of training data in either gs:// uri or
bigquery uri',
        type = str,
        default =  os.environ['AIP_TRAINING_DATA_URI'] if
'AIP_TRAINING_DATA_URI' in os.environ else ""
    )
    parser.add_argument(
        '--validation_data_uri',
        help = 'location of validation data in either gs:// uri or
bigquery uri',
        type = str,
        default =  os.environ['AIP_VALIDATION_DATA_URI'] if
'AIP_VALIDATION_DATA_URI' in os.environ else ""
    )
    parser.add_argument(
        '--test_data_uri',
        help = 'location of test data in either gs:// uri or bigquery
uri',
        type = str,
        default =  os.environ['AIP_TEST_DATA_URI'] if
'AIP_TEST_DATA_URI' in os.environ else ""
    )

    parser.add_argument("-v", "--verbose", help="increase output
verbosity",
                    action="store_true")



    args = parser.parse_args()
    arguments = args.__dict__


    if args.verbose:
        logging.basicConfig(level=logging.INFO)


    logging.info('Model artifacts will be exported here:
{}'.format(arguments['model_dir']))
    logging.info('Data format: {}'.format(arguments["data_format"]))
    logging.info('Training data uri:
{}'.format(arguments['training_data_uri']) )
    logging.info('Validation data uri:
```

```
{}'.format(arguments['validation_data_uri']))
    logging.info('Test data uri:
{}'.format(arguments['test_data_uri']))


    '''
    We have 2 different ways to load our data to pandas. One is from
cloud storage by loading csv files and
    the other is by connecting to BigQuery. Vertex AI supports both
and
    here we created a code that depelnding on the dataset provided, we
will select the appropriated loading method.
    '''
    logging.info('Loading {} data'.format(arguments["data_format"]))
    if(arguments['data_format']=='csv'):
        df_train = load_data_from_gcs(arguments['training_data_uri'])
        df_test = load_data_from_bq(arguments['test_data_uri'])
        df_valid =
load_data_from_gcs(arguments['validation_data_uri'])
    elif(arguments['data_format']=='bigquery'):
        print(arguments['training_data_uri'])
        df_train = load_data_from_bq(arguments['training_data_uri'])
        df_test = load_data_from_bq(arguments['test_data_uri'])
        df_valid = load_data_from_bq(arguments['validation_data_uri'])
    else:
        raise ValueError("Invalid data type ")

    #as we will be using cross validation, we will have just a
training set and a single test set.
    # we ill merge the test and validation to achieve an 80%-20% split
    df_test = pd.concat([df_test,df_valid])

    logging.info('Defining model parameters')
    model_params = dict()
    model_params['kernel'] = arguments['model_param_kernel']
    model_params['degree'] = arguments['model_param_degree']
    model_params['C'] = arguments['model_param_C']
    model_params['probability'] = arguments['model_param_probability']

    df_train = clean_missing_numerics(df_train, NUMERIC_FEATURES)
    df_test = clean_missing_numerics(df_test, NUMERIC_FEATURES)


    logging.info('Running feature selection')
    X_train, y_train = data_selection(df_train, ALL_COLUMNS, LABEL)
    X_test, y_test = data_selection(df_test, ALL_COLUMNS, LABEL)

    logging.info('Training pipelines in CV')
```

```
    clf = pipeline_builder(model_params, BINARY_FEATURES_IDX,
NUMERIC_FEATURES_IDX, CATEGORICAL_FEATURES_IDX)

    cv_score = train_pipeline(clf, X_train, y_train)



    logging.info('Export trained pipeline and report')
    pipeline_export_gcs(clf, arguments['model_dir'])

    y_pred = clf.predict(X_test)


    test_score = f1_score(y_test, y_pred, average='weighted')


    logging.info('f1score: '+ str(test_score))

    report = prepare_report(cv_score,
                            model_params,
                            classification_report(y_test,y_pred),
                            ALL_COLUMNS,
                            X_test.to_numpy()[0:2])

    report_export_gcs(report, arguments['model_dir'])


    logging.info('Training job completed. Exiting...')
```

Build your package

Now it is time to build your package so that you can use it with the training service.

Copy-paste the following code in titanic/setup.py, file is provided in our GitHub repository.

```
from setuptools import find_packages
from setuptools import setup

REQUIRED_PACKAGES = [
    'gcsfs==0.7.1',
    'dask[dataframe]==2021.2.0',
    'google-cloud-bigquery-storage==1.0.0',
    'six==1.15.0'
]

setup(
    name='trainer',
```

```
    version='0.1',
    install_requires=REQUIRED_PACKAGES,
    packages=find_packages(), # Automatically find packages within
this directory or below.
    include_package_data=True, # if packages include any data files,
those will be packed together.
    description='Classification training titanic survivors prediction
model'
)
```

Return to your terminal and test if you can train a model using task.py.

First create the following environment variables but remember to ensure you have selected the right GCP project from the console:

- **PROJECT_ID** Will be set to the selected project id
- **BUCKET_NAME** Will be the PROJECT_ID and "-bucket" attached to it

```
export REGION="europe-west4"
export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
export BUCKET_NAME=$PROJECT_ID"-bucket"
```

First create a bucket where you want to export your trained model.

```
gsutil mb -l $REGION "gs://"$BUCKET_NAME
```

Now run the the following commands. We are using all of our training data to test. Also the same dataset for test, validation and training is used. Here you want to ensure that the code executes and that it is free of bugs. In reality we want to use different test and validation data. We will leave that for Vertex AI training service to handle.

First install the required libraries

```
cd /home/jupyter/titanic
pip install setuptools
python setup.py install
```

Now run your training code to verify that it executes without issues

```
python -m trainer.task -v \
    --model_param_kernel=linear \
    --model_dir="gs://"$BUCKET_NAME"/titanic/trial" \
    --data_format=bigquery \
    --training_data_uri="bq://"$PROJECT_ID".titanic.survivors" \
    --test_data_uri="bq://"$PROJECT_ID".titanic.survivors" \
```

```
    --validation_data_uri="bq://"$PROJECT_ID".titanic.survivors"
```

If the code executed successfully you will be able to see INFO logs printed. The two lines indicate the f1 score which should be around 0.85 and the last line idicating that the training job completed successfully:

```
INFO:root:f1score: 0.85
INFO:root:Training job completed. Exiting...
```

Cogratulations! You are ready to create your training python package!

The following command does exactly that:

```
cd /home/jupyter/titanic
python setup.py sdist
```

After the command executes you will see a new folder called dist that contains a tar.gz file. This is your python package

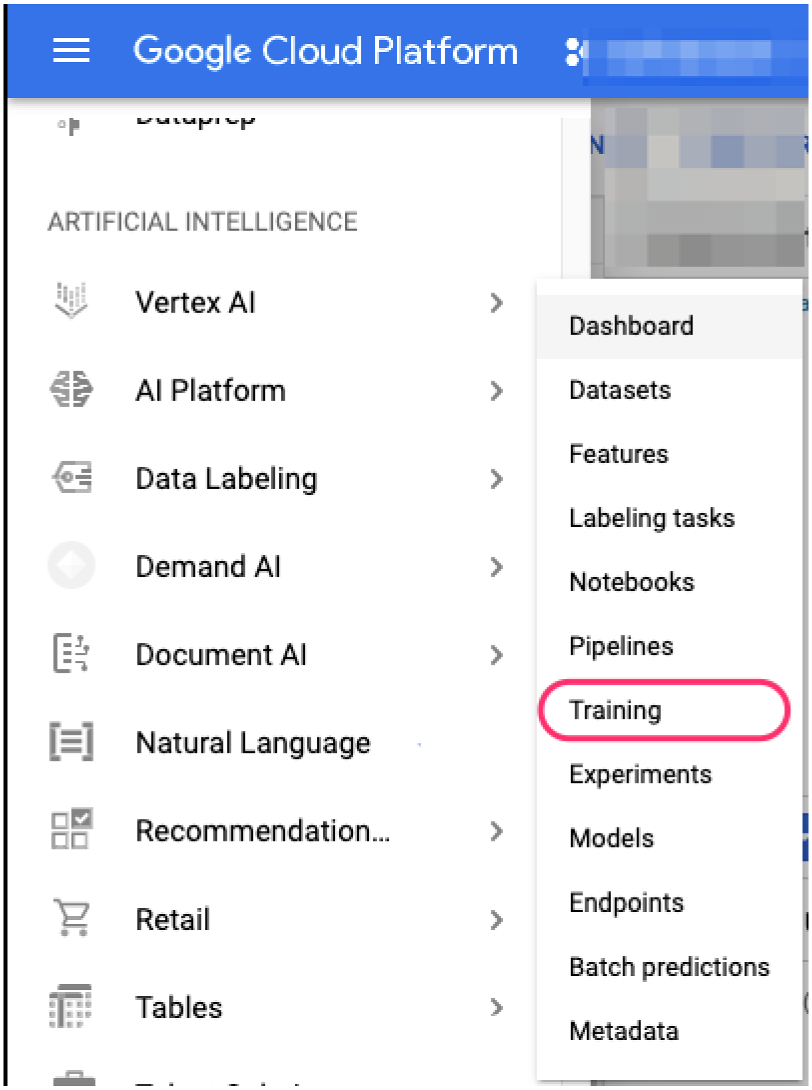
You should copy the package to GCS so that the training service can use it to train a new model when you need to

```
gsutil cp dist/trainer-0.1.tar.gz
"gs://"$BUCKET_NAME"/titanic/dist/trainer-0.1.tar.gz"
```

Model Training

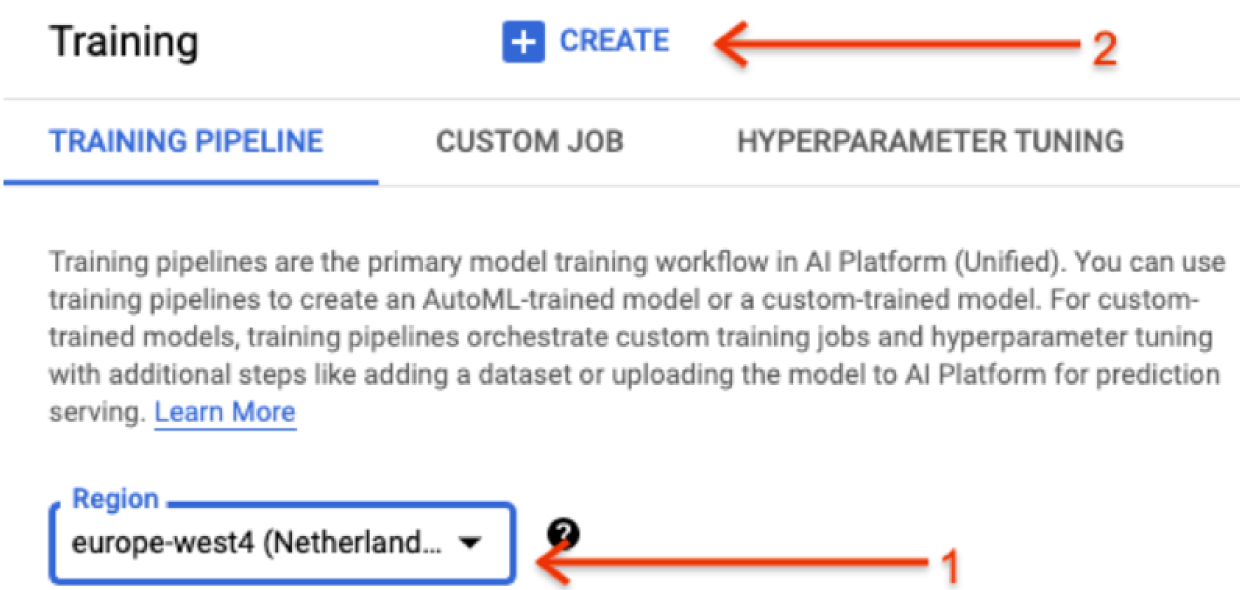
In this section you will train a model on Vertex AI. You are going to use the GUI for that. There is also a programmatic way using a python SDK, however using the GUI helps to better understand the process.

From the Google Cloud console navigate to Vertex AI -> Training



Step 0:

Select the region as europe-west4 and click create as the picture below:

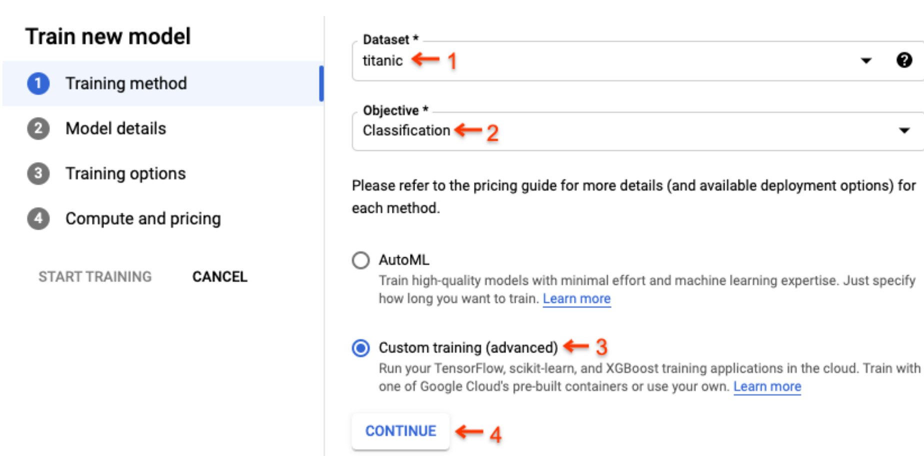


Step 1: Training method

In this step select the dataset and define the objective for the training job.

1. **Dataset**: The dataset we created few steps back. The name should be **titanic**
2. **Objective**: The model predicts if an individual is likely to survive the titanic tragedy or not. This is a **Classification** problem
3. **Custom Training**: You want to use your custom training package.

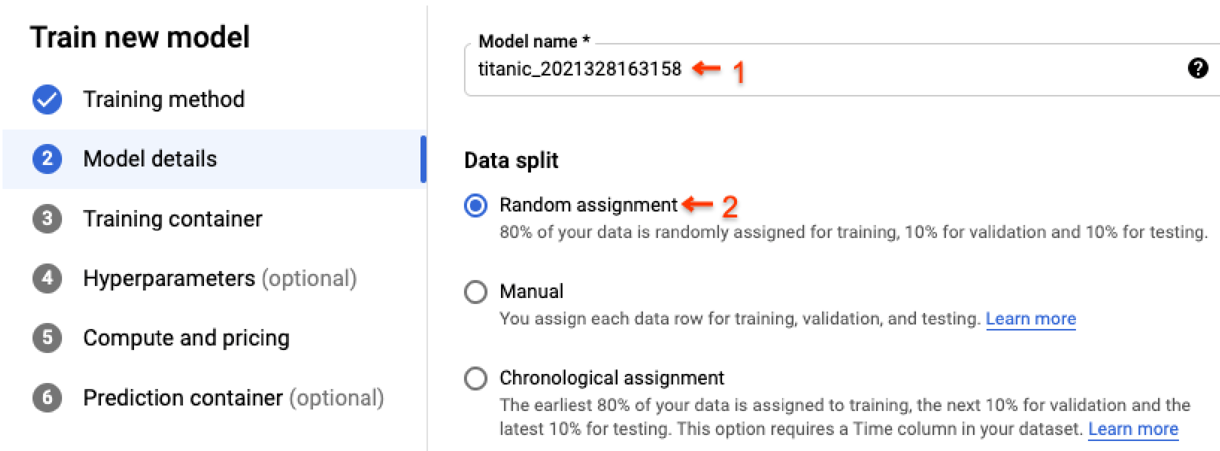
Click **CONTINUE**



Step 2: Model details

Now define the model name. The default name should be the name of the dataset and a timestamp. You can leave it as is. If you click **show more** you will see the option to define the split of data into training, test and validation sets. Random assignment will randomly split the data into training, validation and testing. This seems like a good option.

Click **CONTINUE**



Step 3: Training container

Define your training environment.

1. **Pre-built container:** Google cloud offers a set of prebuilt containers that make it easy to train your models. Those containers support frameworks such as Scikit-Learn, Tensorflow and XGBoost. If your training job is using something exotic you will need to prepare and provide a container for training(custom container). Your model is based on scikit-learn and prebuilt container already exists.
2. **Model framework:** Scikit-learn. This is the library you used for model training.
3. **Model framework version:** Your code is compatible with 0.23.
4. **Package location:** You can browse to the location of your training package. This is the location where you uploaded training-0.1.tar.gz. If you followed the previous steps correctly the location should be gs://YOUR-BUCKET-NAME/titanic/dist/trainer-0.1.tar.gz and YOUR-BUCKET-NAME is the name of the bucket you used under the *Build your package* section
5. **Python Module:** The python module you created in Notebooks. It will correspond to the folder that has your training code/module and the name of the entry file. This should be trainer.task
6. **BigQuey project for exporting data:** In step 1 you selected the dataset and defined automatic split. A new dataset and tables for train/test/validate sets will be created under the selected project. Select the same project you are running the lab.

Additionally training/test/validation datasets URIs will be set as environment variables in the training container so You can automatically use those variables to load your data. The environment variable names for the datasets will be AIP_TRAINING_DATA_URI, AIP_TEST_DATA_URI, AIP_VALIDATION_DATA_URI. An additional variable will be AIP_DATA_FORMAT which will be either csv or bigquery, depending on the type of the selected dataset in Step 1. You have already built this logic in *task.py* . Observe this example code (taken from task.py):

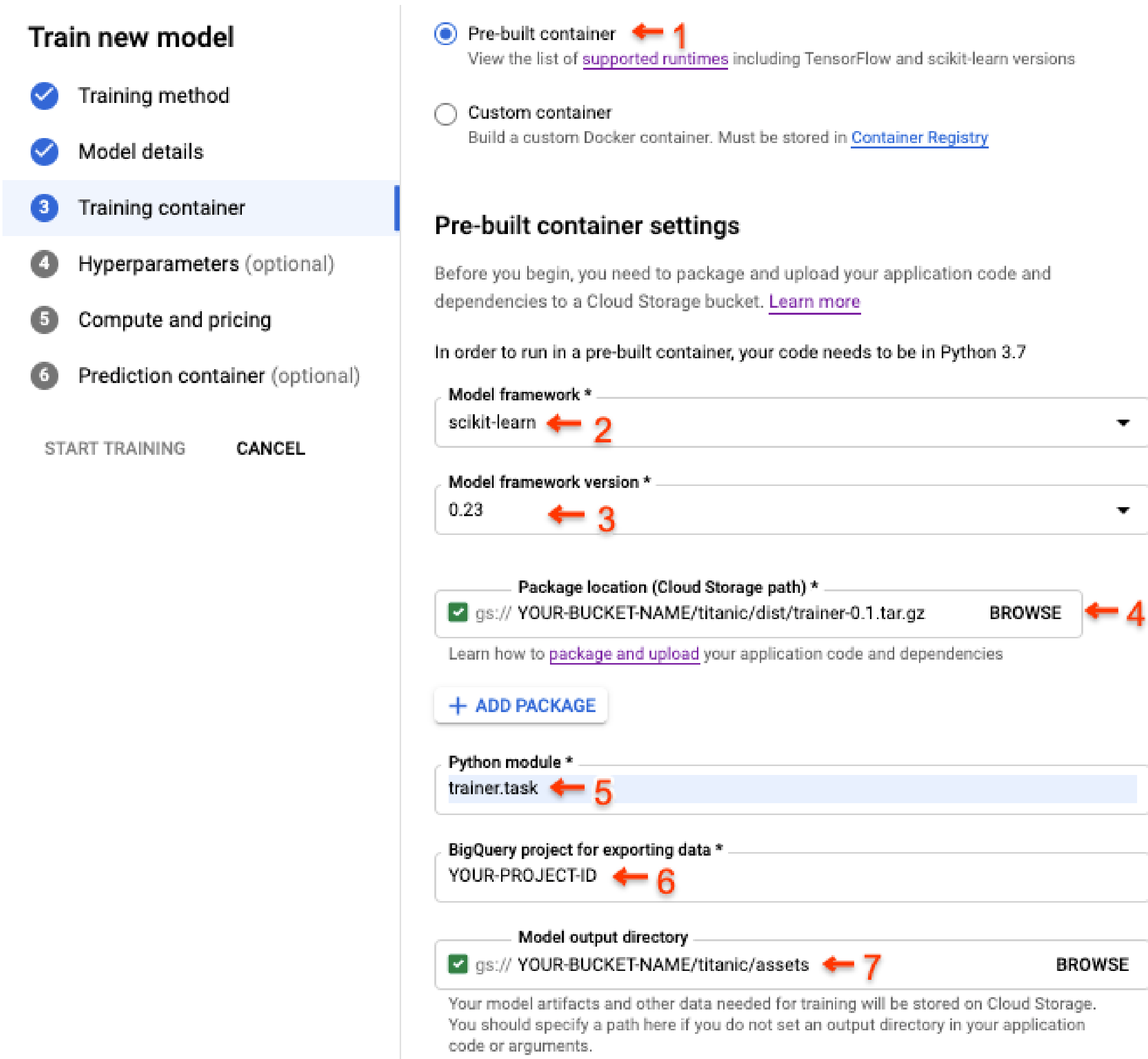
```
...
parser.add_argument( '--training_data_uri ',
   help = 'Directory to output model and artifacts',
   type = str,
   default = os.environ['AIP_TRAINING_DATA_URI'] if 'AIP_TRAINING_DATA_URI' in
os.environ else ""  )
...
```

7. **Model output directory:** The location the model will be exported to. This is going to be an environment variable in the training container called AIP_MODEL_DIR. In our *task.py* there is an input parameters to capture this:

```
...
parser.add_argument( '--model_dir',
        help = 'Directory to output model and artifacts',
       type = str,
       default = os.environ['AIP_MODEL_DIR'] if 'AIP_MODEL_DIR' in os.environ else
""  )
...
```

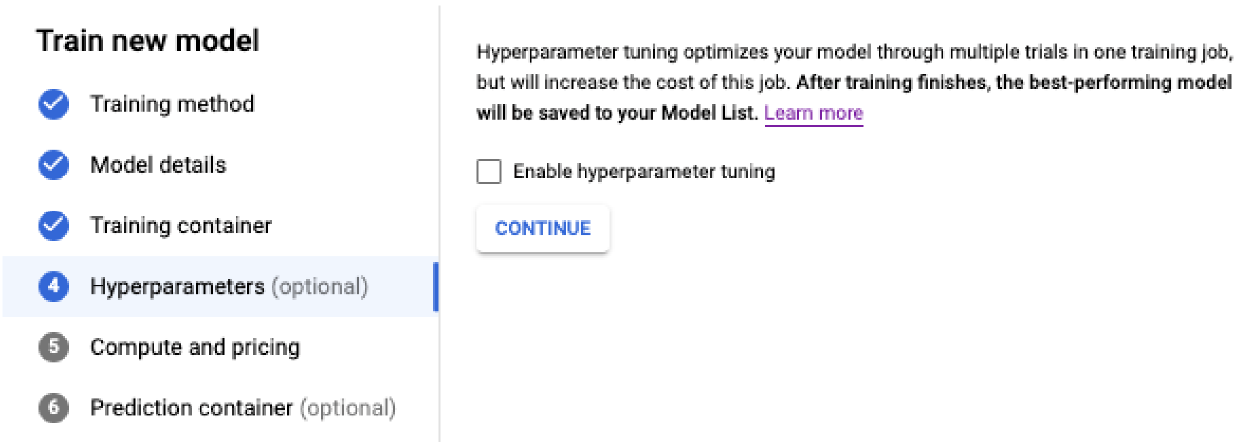
You can use the environment variable to know where to export the training job artifacts. Let's select: gs://YOUR-BUCKET-NAME/training/assets

Click **CONTINUE**



Step 4: Hyperparameter tuning

Hyperparameter tuning section allows you to define a set of model parameters that you would like to tune your model with. Different values will be explored in order to produce the model with the best parameters. In our code we did not implement the hyperparameter tuner functionality. It's only a few lines of code (about 5) but we did not want to add this complexity now. Let's skip this step by pressing **CONTINUE**

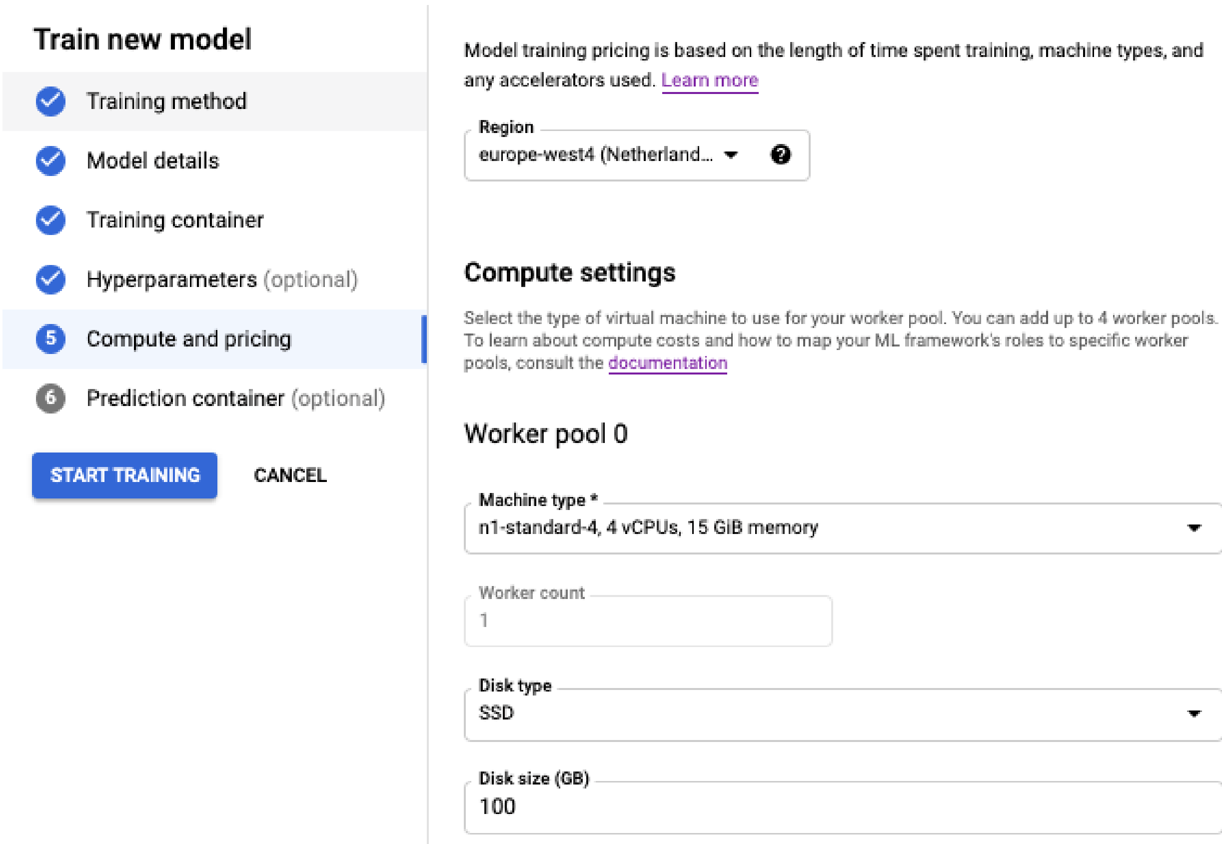


Step 5: Compute and pricing

Where do we want our training job to run and what type of server do we want to use? Your model training process is not hungry for resources. We were able to run the training job inside a relatively small notebook instance and the execution finishes quite fast. With that in mind we choose:

- **Region:** europe-west4
- **Machine type:** n1-standard-4

Click **CONTINUE**



Step 6: Prediction container

In this step you can decide if you want to just train the model or also add add settings for the prediction service used to productionise your model.

You will be using a pre-built container in this lab, however keep in mind that Vertex AI gives you a few options for model serving:

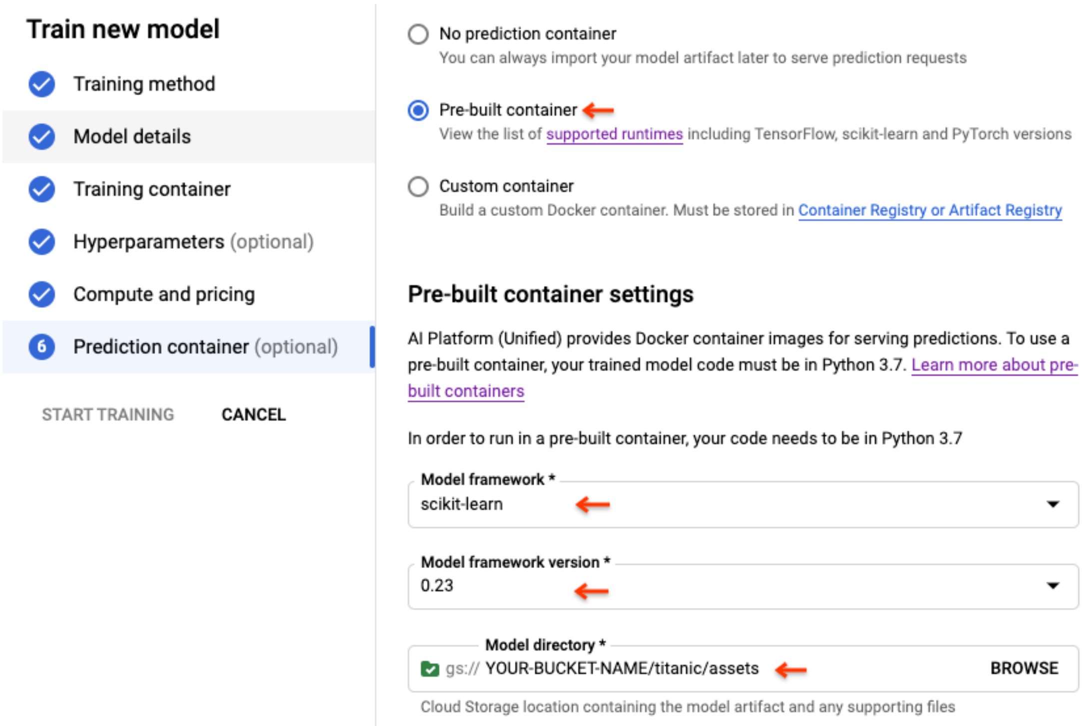
- **No Prediction Container:** Just train the model and worry about productionizing the model later
- **Pre-built container:** Train the model and define the prebuilt container to be used for deployment
- **Custom container:** Train the model and define a custom container to be used for deployment

You should choose a Pre-built container since Google Cloud already offers a Scikit-Learn container. You will deploy the model after the training job is completed.

- **Model framework:** scikit-learn
- **Model framework version:** 0.23

- **Model directory:** gs://YOUR-BUCKET-NAME/training/assets This should be the same as the model output directory you defined in step 3

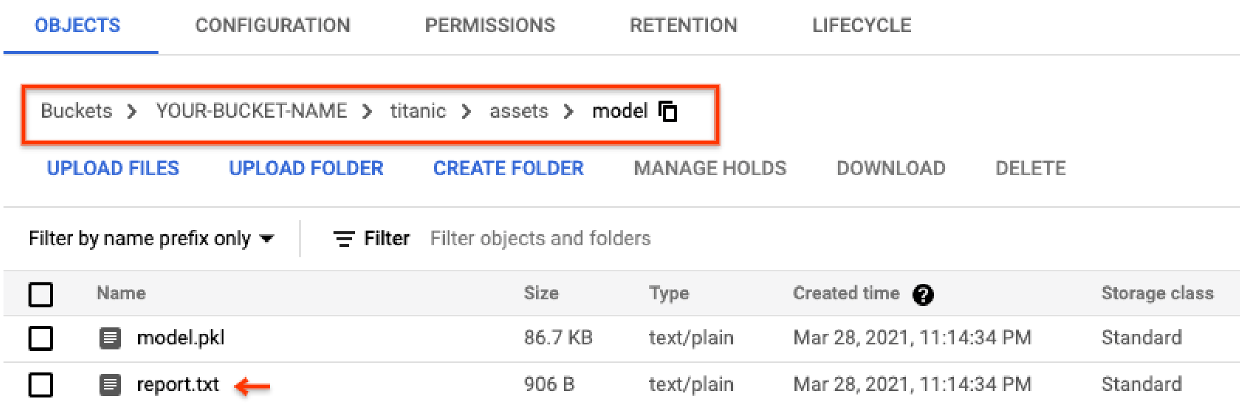
Click **START TRAINING**



The new training job will show under the **TRAINING PIPELINE** tab. The training will take about 17 minutes to complete

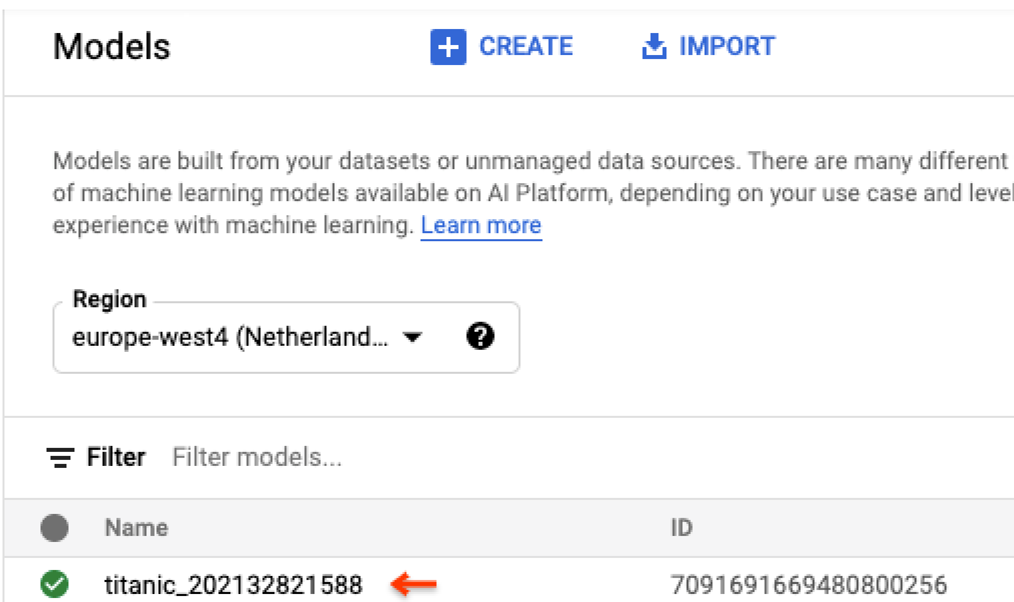
Model Evaluation

After the training job completion artifacts will be exported under gs://YOUR-BUCKET-NAME/training/assets You can inspect the report.txt file which contains evaluation metrics and classification report of the model.



Model Deployment

Last step is model deployment! After the model training job is completed (just under 20 minutes), select the trained model and deploy it to an endpoint.



Click on the trained model and **DEPLOY TO ENDPOINT**

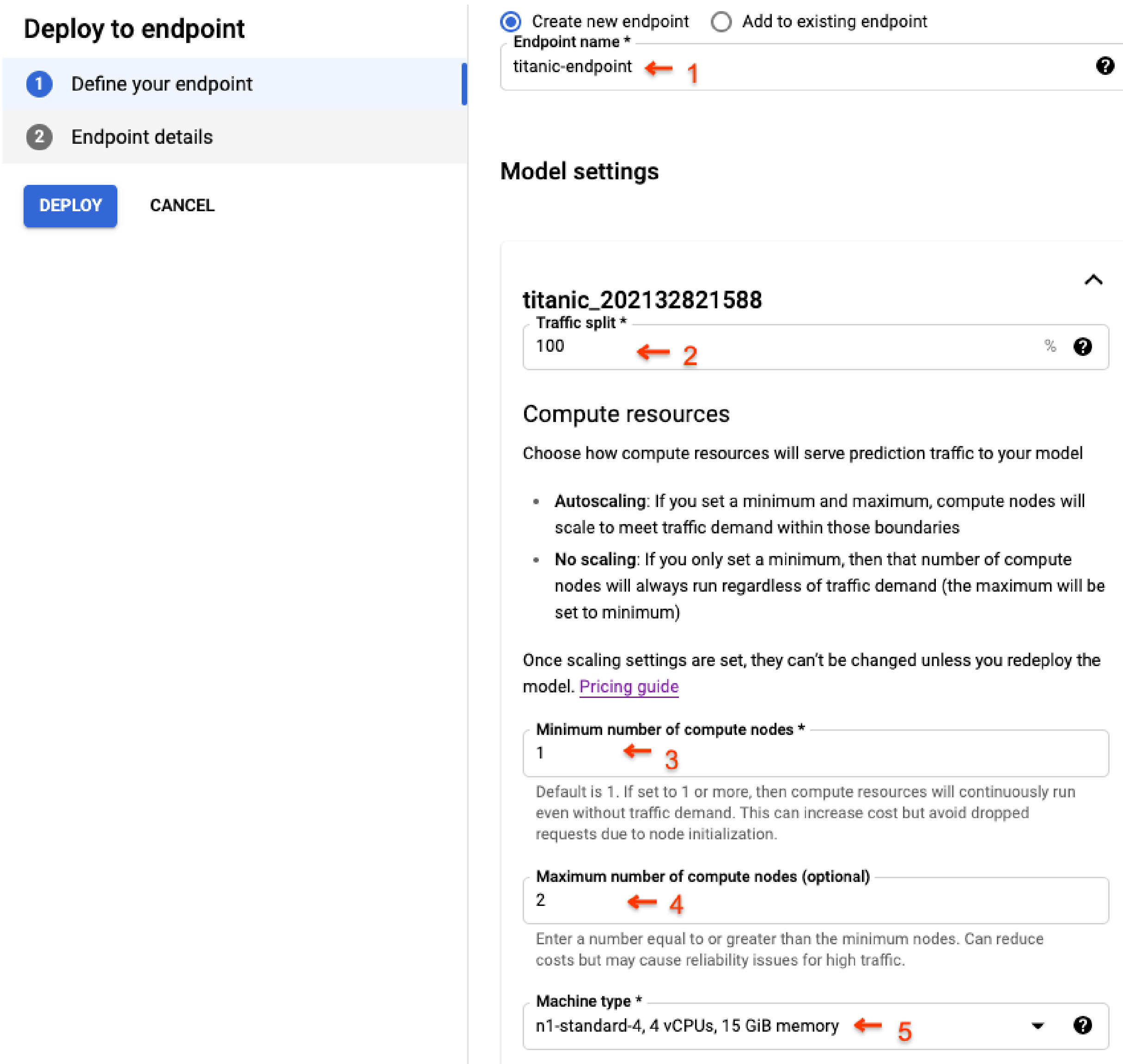
On the popup you can define the required resources for model deployment:

1. **Endpoint name:** Endpoint URL where the model is served. A reasonable name for that would be titanic-endpoint
2. **Traffic split:** Defines the percentage of traffic that you want to direct to this model. An endpoint can have multiple models and you can despite how to split

the traffic among them. In this case you are deploying a singe model so the traffic has to be 100 percent.

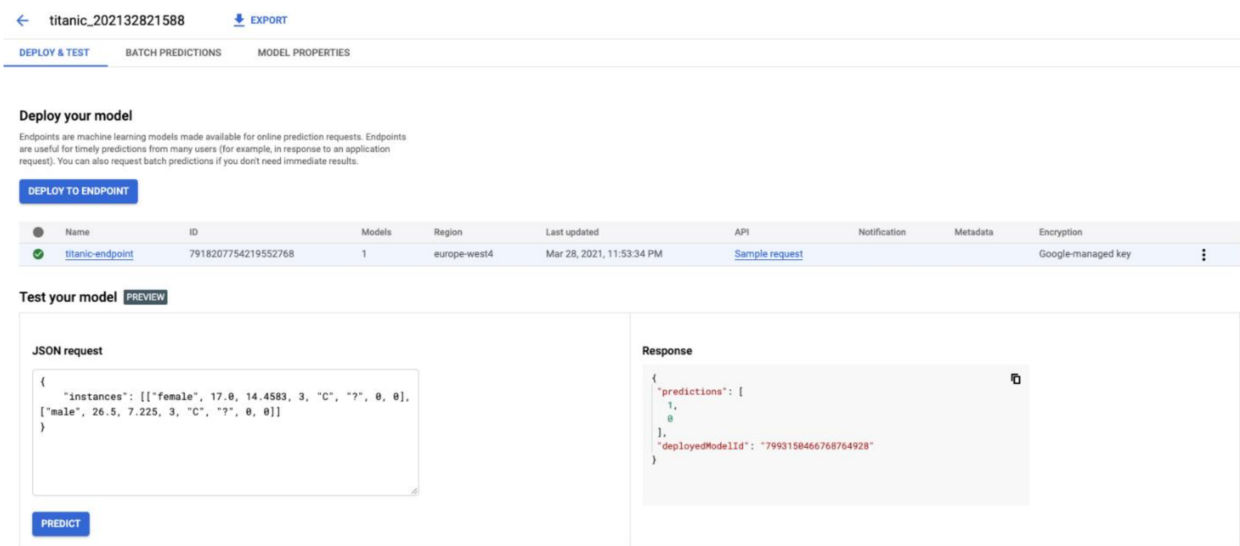
3. **Minimum number of compute nodes:** The minimum number of nodes required to serve model predictions. Start with 1. Additionally the prediction service will autoscale in case there is traffic
4. **Maximum number of compute nodes:** In case of autoscaling, this variable defines the upper limit of nodes. It helps protecting against unwanted costs that autoscaling might result in. Set this variable to 2
5. **Machine type:** Google cloud offers a set of machine types you can deploy your model to. Each machine has its own memory and vcpu specs. Your model is simple so serving on an n1-standard-4 instance will do the job

Click **CONTINUE** and **DEPLOY**



Model Prediction

Under **Models** test the model prediction endpoint. The GUI provides a form to send a json request payload and responds back with the predictions as well as the model id used for the prediction. That is because you can deploy more than one model to an endpoint and split the traffic.



Try the following payload and perhaps change some of the values to see how the predictions change: The sequence of the input features is [‘sex', ‘age', ‘fare', ‘pclass', ‘embarked', ‘home_dest', ‘parch', ‘sibsp']

```
{
    "instances": [
      ["male", 29.8811345124283, 26.0, 1, "S", "New York, NY", 0, 0],
      ["female", 48.0, 39.6, 1, "C", "London / Paris", 0, 1]]
}
```

The endpoint responds with a list of Zeros or Ones in the same order as your input. 0 means it is more likely that the individual will not survive the titanic accident and 1 means the individual is likely to survive it.

Clean up

We should do this in our temporary experiment account, but you may wish to take down your Dataset and Notebook if you're using your own project in the future to be spend efficient.

Delete the dataset in Big Query

You can delete the dataset in Big Query.

Remove the endpoint and model

In the Vertex AI Model, remove the endpoint and then remove the model.

Delete the AI Notebook instance

If you ran the "Notebook" part of the experiment, you can **DELETE** or **STOP** the notebook instance from the Cloud Console.

Congratulations!

In this experiment, you created and ran an ML workflow learning model focused on the Titanic maritime tragedy.

Titanic Data Lineage and Citation

The original Titanic dataset, describing the survival status of individual passengers on the Titanic. The titanic data does not contain information from the crew, but it does contain actual ages of half of the passengers. The principal source for data about Titanic passengers is the Encyclopedia Titanica.

The datasets used here were begun by a variety of researchers. One of the original sources is Eaton & Haas (1994) Titanic: Triumph and Tragedy, Patrick Stephens Ltd, which includes a passenger list created by many researchers and edited by Michael A. Findlay.

Thomas Cason of UVa has greatly updated and improved the titanic data frame using the Encyclopedia Titanica and created the dataset here. Some duplicate passengers have been dropped, many errors corrected, many missing ages filled in, and new variables created.

For more information about how this dataset was constructed:
http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3info.txt