

Logistic regression for binary classification with Core APIs

Run in
 [Google](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/core) (<https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/core>)
[Colab](#)

This guide demonstrates how to use the [TensorFlow Core low-level APIs](https://www.tensorflow.org/guide/core) (<https://www.tensorflow.org/guide/core>) to perform [binary classification](https://developers.google.com/machine-learning/glossary#binary_classification) (https://developers.google.com/machine-learning/glossary#binary_classification) with [logistic regression](https://developers.google.com/machine-learning/crash-course/logistic-regression/) (<https://developers.google.com/machine-learning/crash-course/logistic-regression/>). It uses the [Wisconsin Breast Cancer Dataset](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)) ([https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+\(original\)](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original))) for tumor classification.

[Logistic regression](https://developers.google.com/machine-learning/crash-course/logistic-regression/) (<https://developers.google.com/machine-learning/crash-course/logistic-regression/>) is one of the most popular algorithms for binary classification. Given a set of examples with features, the goal of logistic regression is to output values between 0 and 1, which can be interpreted as the probabilities of each example belonging to a particular class.

Setup

This tutorial uses [pandas](https://pandas.pydata.org) (<https://pandas.pydata.org>) for reading a CSV file into a [DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html) (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html>), [seaborn](https://seaborn.pydata.org) (<https://seaborn.pydata.org>) for plotting a pairwise relationship in a dataset, [Scikit-learn](https://scikit-learn.org/) (<https://scikit-learn.org/>) for computing a confusion matrix, and [matplotlib](https://matplotlib.org/) (<https://matplotlib.org/>) for creating visualizations.

```
$ pip install -q seaborn
```

```
import tensorflow as tf
import pandas as pd
import matplotlib
from matplotlib import pyplot as plt
import seaborn as sns
import sklearn.metrics as sk_metrics
```

```

import tempfile
import os

# Preset matplotlib figure sizes.
matplotlib.rcParams['figure.figsize'] = [9, 6]

print(tf.__version__)
# To make the results reproducible, set the random seed value.
tf.random.set_seed(22)

2024-08-15 02:45:41.468739: E external/local_xla/xla/stream_executor/cuda/cuda_
2024-08-15 02:45:41.489749: E external/local_xla/xla/stream_executor/cuda/cuda_
2024-08-15 02:45:41.496228: E external/local_xla/xla/stream_executor/cuda/cuda_
2.17.0

```

Load the data

Next, load the Wisconsin Breast Cancer Dataset

([https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+\(original\)](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original))) from the UCI Machine Learning Repository (<https://archive.ics.uci.edu/ml/>). This dataset contains various features such as a tumor's radius, texture, and concavity.

```

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wi
features = ['radius', 'texture', 'perimeter', 'area', 'smoothness', 'compactnes
            'concavity', 'concave_points', 'symmetry', 'fractal_dimension']
column_names = ['id', 'diagnosis']

for attr in ['mean', 'std', 'largest']:
    for feature in features:
        column_names.append(feature + "_" + attr)

```

Read the dataset into a pandas DataFrame () using pandas.read_csv

(https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html):

```

dataset = pd.read_csv(url, names=column_names)

```

```
dataset.info()
```

```
20  symmetry_ste          569 non-null    float64
21  fractal_dimension_ste  569 non-null    float64
22  radius_largest        569 non-null    float64
23  texture_largest       569 non-null    float64
24  perimeter_largest     569 non-null    float64
25  area_largest          569 non-null    float64
26  smoothness_largest    569 non-null    float64
27  compactness_largest   569 non-null    float64
28  concavity_largest     569 non-null    float64
29  concave_poinits_largest 569 non-null    float64
30  symmetry_largest      569 non-null    float64
31  fractal_dimension_largest 569 non-null    float64
dtypes: float64(30), int64(1), object(1)
memory usage: 142.4+ KB
```

Display the first five rows:

```
dataset.head()
```

	iddiagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_r
42302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760
42517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864

Split the dataset into training and test sets using **pandas.DataFrame.sample**

(<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html>),

pandas.DataFrame.drop

(<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop.html>) and

pandas.DataFrame.iloc (<https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html>)

. Make sure to split the features from the target labels. The test set is used to evaluate your model's generalizability to unseen data.

```
train_dataset = dataset.sample(frac=0.75, random_state=1)
```

```
len(train_dataset)
```

427

```
test_dataset = dataset.drop(train_dataset.index)
```

```
len(test_dataset)
```

142

```
# The `id` column can be dropped since each row is unique
x_train, y_train = train_dataset.iloc[:, 2:], train_dataset.iloc[:, 1]
x_test, y_test = test_dataset.iloc[:, 2:], test_dataset.iloc[:, 1]
```

Preprocess the data

This dataset contains the mean, standard error, and largest values for each of the 10 tumor measurements collected per example. The "diagnosis" target column is a categorical variable with 'M' indicating a malignant tumor and 'B' indicating a benign tumor diagnosis. This column needs to be converted into a numerical binary format for model training.

The [pandas.Series.map](https://pandas.pydata.org/docs/reference/api/pandas.Series.map.html) (<https://pandas.pydata.org/docs/reference/api/pandas.Series.map.html>) function is useful for mapping binary values to the categories.

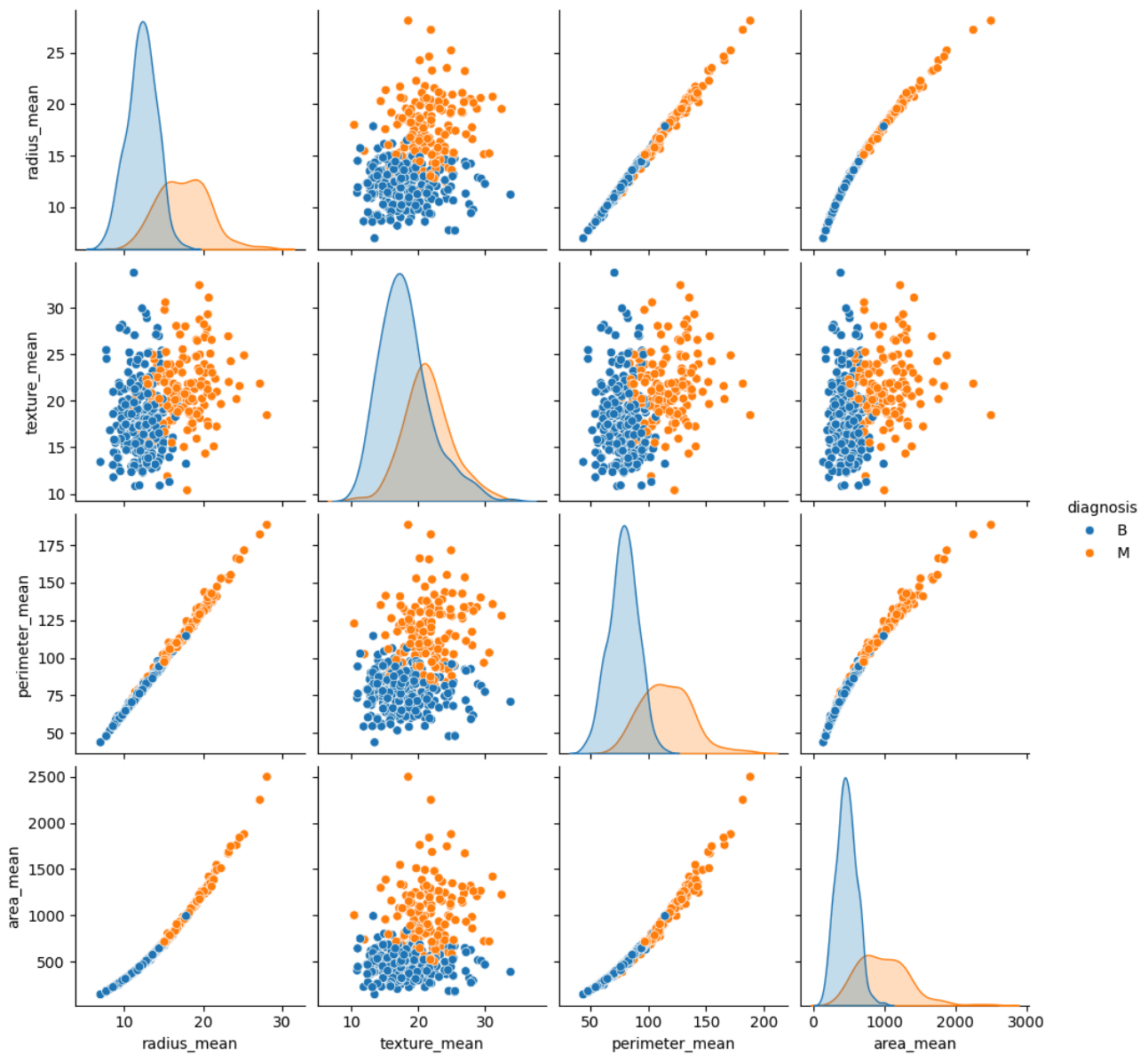
The dataset should also be converted to a tensor with the [tf.convert_to_tensor](https://www.tensorflow.org/api_docs/python/tf/convert_to_tensor) (https://www.tensorflow.org/api_docs/python/tf/convert_to_tensor) function after the preprocessing is complete.

```
y_train, y_test = y_train.map({'B': 0, 'M': 1}), y_test.map({'B': 0, 'M': 1})
x_train, y_train = tf.convert_to_tensor(x_train, dtype=tf.float32), tf.convert_to_
x_test, y_test = tf.convert_to_tensor(x_test, dtype=tf.float32), tf.convert_to_
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to
I0000 00:00:1723689945.265757 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.269593 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.273290 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.276976 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.288712 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.292180 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.295550 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.299093 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.302584 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.306098 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.309484 132290 cuda_executor.cc:1015] successful NUMA n
I0000 00:00:1723689945.312921 132290 cuda_executor.cc:1015] successful NUMA n
```

Use [seaborn.pairplot](https://seaborn.pydata.org/generated/seaborn.pairplot.html) (<https://seaborn.pydata.org/generated/seaborn.pairplot.html>) to review the joint distribution of a few pairs of mean-based features from the training set and observe how they relate to the target:

```
sns.pairplot(train_dataset.iloc[:, 1:6], hue = 'diagnosis', diag_kind='kde');
```



This pairplot demonstrates that certain features such as radius, perimeter and area are highly correlated. This is expected since the tumor radius is directly involved in the computation of both perimeter and area. Additionally, note that malignant diagnoses seem to be more right-skewed for many of the features.

Make sure to also check the overall statistics. Note how each feature covers a vastly different range of values.

```
train_dataset.describe().transpose()[10]
```

	count	mean	std	min	25%	50%
	427	0.2756014e+07	1.162735e+08	670.00000865427	500000905539.000008	8108296
..	427	0.1414331e+01	3.528717e+00	6.98100 11.695000	13.43000	1.594000e

Given the inconsistent ranges, it is beneficial to standardize the data such that each feature has a zero mean and unit variance. This process is called normalization (<https://developers.google.com/machine-learning/glossary#normalization>).

```
class Normalize(tf.Module):
    def __init__(self, x):
        # Initialize the mean and standard deviation for normalization
        self.mean = tf.Variable(tf.math.reduce_mean(x, axis=0))
        self.std = tf.Variable(tf.math.reduce_std(x, axis=0))

    def norm(self, x):
        # Normalize the input
        return (x - self.mean)/self.std

    def unnorm(self, x):
        # Unnormalize the input
        return (x * self.std) + self.mean

norm_x = Normalize(x_train)
x_train_norm, x_test_norm = norm_x.norm(x_train), norm_x.norm(x_test)
```

Logistic regression

Before building a logistic regression model, it is crucial to understand the method's differences compared to traditional linear regression.

Logistic regression fundamentals

Linear regression returns a linear combination of its inputs; this output is unbounded. The output of a logistic regression (https://developers.google.com/machine-learning/glossary#logistic_regression) is in the $(0, 1)$ range. For each example, it represents the probability that the example belongs to the *positive* class.

Logistic regression maps the continuous outputs of traditional linear regression, $(-\infty, \infty)$, to probabilities, $(0, 1)$. This transformation is also symmetric so that flipping the sign of the

linear output results in the inverse of the original probability.

Let Y denote the probability of being in class 1 (the tumor is malignant). The desired mapping can be achieved by interpreting the linear regression output as the log odds

(<https://developers.google.com/machine-learning/glossary#log-odds>) ratio of being in class 1 as opposed to class 0:

$$\ln\left(\frac{Y}{1-Y}\right) = wX + b$$

By setting $wX + b = z$, this equation can then be solved for Y :

$$Y = \frac{e^z}{1 + e^z} = \frac{1}{1 + e^{-z}}$$

The expression $\frac{1}{1+e^{-z}}$ is known as the sigmoid function

(https://developers.google.com/machine-learning/glossary#sigmoid_function) $\sigma(z)$. Hence, the equation for logistic regression can be written as $Y = \sigma(wX + b)$.

The dataset in this tutorial deals with a high-dimensional feature matrix. Therefore, the above equation must be rewritten in a matrix vector form as follows:

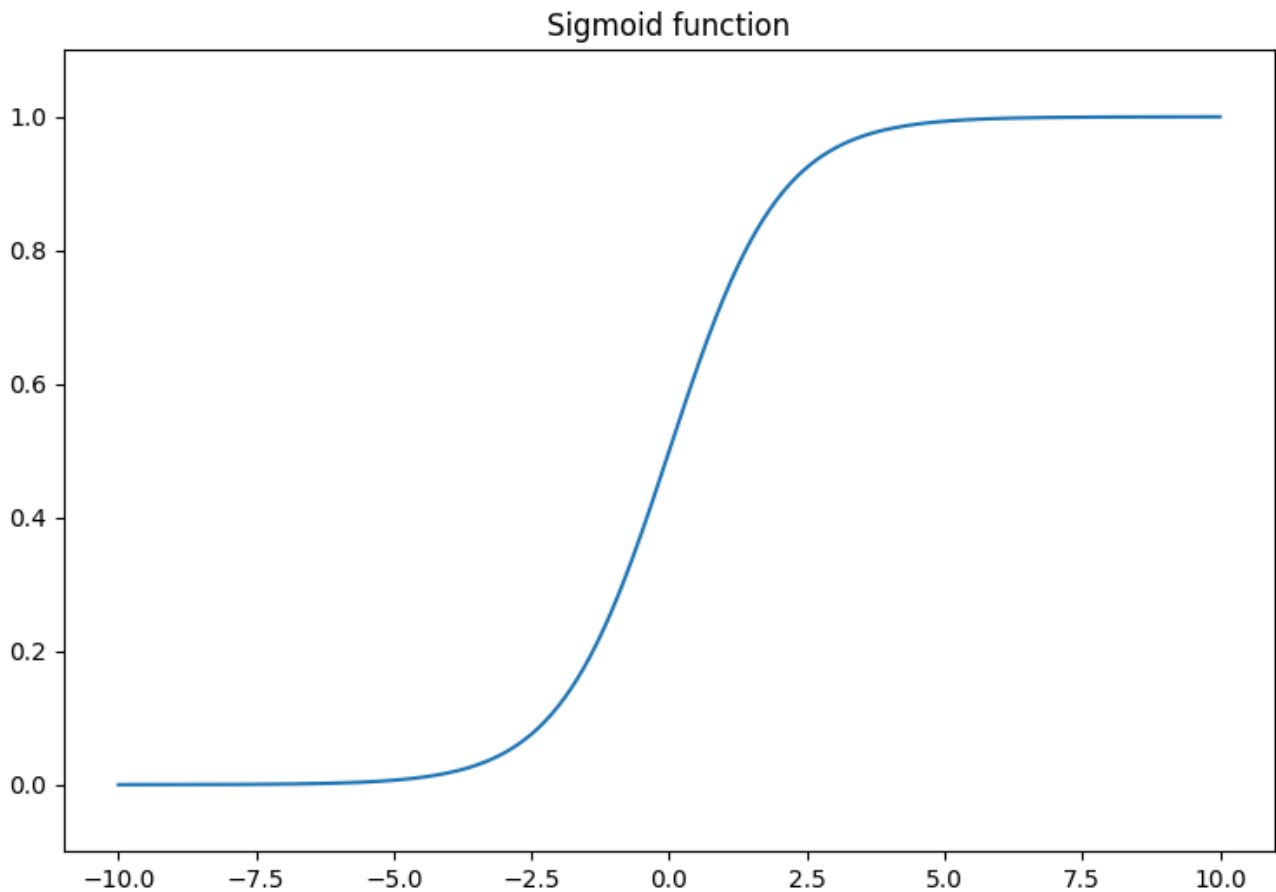
$$Y = \sigma(Xw + b)$$

where:

- Y : a target vector
 $m \times 1$
- X : a feature matrix
 $m \times n$
- w : a weight vector
 $n \times 1$
- b : a bias
- σ : a sigmoid function applied to each element of the output vector

Start by visualizing the sigmoid function, which transforms the linear output, $(-\infty, \infty)$, to fall between 0 and 1. The sigmoid function is available in `tf.math.sigmoid`.

```
x = tf.linspace(-10, 10, 500)
x = tf.cast(x, tf.float32)
f = lambda x : (1/20)*x + 0.6
plt.plot(x, tf.math.sigmoid(x))
plt.ylim((-0.1,1.1))
plt.title("Sigmoid function");
```

The log loss function

The log loss (https://developers.google.com/machine-learning/glossary#Log_Loss), or binary cross-entropy loss, is the ideal loss function for a binary classification problem with logistic regression. For each example, the log loss quantifies the similarity between a predicted probability and the example's true value. It is determined by the following equation:

$$L = -\frac{1}{m} \sum_{i=1}^m y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

where:

- \hat{y} : a vector of predicted probabilities
- y : a vector of true targets

You can use the `tf.nn.sigmoid_cross_entropy_with_logits`

(https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits) function to compute the log loss. This function automatically applies the sigmoid activation to the regression output:

```
def log_loss(y_pred, y):
    # Compute the log loss function
    ce = tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=y_pred)
    return tf.reduce_mean(ce)
```

The gradient descent update rule

The TensorFlow Core APIs support automatic differentiation with `tf.GradientTape` (https://www.tensorflow.org/api_docs/python/tf/GradientTape). If you are curious about the mathematics behind the logistic regression [gradient updates](https://developers.google.com/machine-learning/glossary#gradient_descent) (https://developers.google.com/machine-learning/glossary#gradient_descent), here is a short explanation:

In the above equation for the log loss, recall that each \hat{y}_i can be rewritten in terms of the inputs as $\sigma(\mathbf{X}_i \mathbf{w} + b)$.

The goal is to find a \mathbf{w}^* and b^* that minimize the log loss:

$$L = -\frac{1}{m} \sum_{i=1}^m y_i \cdot \log(\sigma(\mathbf{X}_i \mathbf{w} + b)) + (1 - y_i) \cdot \log(1 - \sigma(\mathbf{X}_i \mathbf{w} + b))$$

By taking the gradient L with respect to \mathbf{w} , you get the following:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{1}{m} (\sigma(\mathbf{X} \mathbf{w} + b) - \mathbf{y}) \mathbf{X}$$

By taking the gradient L with respect to b , you get the following:

$$\frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^m \sigma(\mathbf{X}_i \mathbf{w} + b) - y_i$$

Now, build the logistic regression model.

```
class LogisticRegression(tf.Module):

    def __init__(self):
        self.built = False

    def __call__(self, x, train=True):
        # Initialize the model parameters on the first call
        if not self.built:
            # Randomly generate the weights and the bias term
```

```

    rand_w = tf.random.uniform(shape=[x.shape[-1], 1], seed=22)
    rand_b = tf.random.uniform(shape=[], seed=22)
    self.w = tf.Variable(rand_w)
    self.b = tf.Variable(rand_b)
    self.built = True
# Compute the model output
z = tf.add(tf.matmul(x, self.w), self.b)
z = tf.squeeze(z, axis=1)
if train:
    return z
return tf.sigmoid(z)

```

To validate, make sure the untrained model outputs values in the range of (0, 1) for a small subset of the training data.

```
log_reg = LogisticRegression()
```

```

y_pred = log_reg(x_train_norm[:5], train=False)
y_pred.numpy()

```

```

array([0.9994985 , 0.9978607 , 0.29620072, 0.01979049, 0.3314926 ],
      dtype=float32)

```

Next, write an accuracy function to calculate the proportion of correct classifications during training. In order to retrieve the classifications from the predicted probabilities, set a threshold for which all probabilities higher than the threshold belong to class 1. This is a configurable hyperparameter that can be set to 0.5 as a default.

```

def predict_class(y_pred, thresh=0.5):
    # Return a tensor with `1` if `y_pred` > `0.5`, and `0` otherwise
    return tf.cast(y_pred > thresh, tf.float32)

def accuracy(y_pred, y):
    # Return the proportion of matches between `y_pred` and `y`
    y_pred = tf.math.sigmoid(y_pred)
    y_pred_class = predict_class(y_pred)
    check_equal = tf.cast(y_pred_class == y, tf.float32)
    acc_val = tf.reduce_mean(check_equal)

```

```
return acc_val
```

Train the model

Using mini-batches for training provides both memory efficiency and faster convergence. The `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) API has useful functions for batching and shuffling. The API enables you to build complex input pipelines from simple, reusable pieces.

```
batch_size = 64
train_dataset = tf.data.Dataset.from_tensor_slices((x_train_norm, y_train))
train_dataset = train_dataset.shuffle(buffer_size=x_train.shape[0]).batch(batch_size)
test_dataset = tf.data.Dataset.from_tensor_slices((x_test_norm, y_test))
test_dataset = test_dataset.shuffle(buffer_size=x_test.shape[0]).batch(batch_size)
```

Now write a training loop for the logistic regression model. The loop utilizes the log loss function and its gradients with respect to the input in order to iteratively update the model's parameters.

```
# Set training parameters
epochs = 200
learning_rate = 0.01
train_losses, test_losses = [], []
train_accs, test_accs = [], []

# Set up the training loop and begin training
for epoch in range(epochs):
    batch_losses_train, batch_accs_train = [], []
    batch_losses_test, batch_accs_test = [], []

    # Iterate over the training data
    for x_batch, y_batch in train_dataset:
        with tf.GradientTape() as tape:
            y_pred_batch = log_reg(x_batch)
            batch_loss = log_loss(y_pred_batch, y_batch)
            batch_acc = accuracy(y_pred_batch, y_batch)
        # Update the parameters with respect to the gradient calculations
        grads = tape.gradient(batch_loss, log_reg.variables)
        for g, v in zip(grads, log_reg.variables):
            v.assign_sub(learning_rate * g)
    # Keep track of batch-level training performance
    batch_losses_train.append(batch_loss)
```

```

batch_accs_train.append(batch_acc)

# Iterate over the testing data
for x_batch, y_batch in test_dataset:
    y_pred_batch = log_reg(x_batch)
    batch_loss = log_loss(y_pred_batch, y_batch)
    batch_acc = accuracy(y_pred_batch, y_batch)
    # Keep track of batch-level testing performance
    batch_losses_test.append(batch_loss)
    batch_accs_test.append(batch_acc)

# Keep track of epoch-level model performance
train_loss, train_acc = tf.reduce_mean(batch_losses_train), tf.reduce_mean(batch_accs_train)
test_loss, test_acc = tf.reduce_mean(batch_losses_test), tf.reduce_mean(batch_accs_test)
train_losses.append(train_loss)
train_accs.append(train_acc)
test_losses.append(test_loss)
test_accs.append(test_acc)
if epoch % 20 == 0:
    print(f"Epoch: {epoch}, Training log loss: {train_loss:.3f}")

```

```

Epoch: 0, Training log loss: 0.661
Epoch: 20, Training log loss: 0.418
Epoch: 40, Training log loss: 0.269
Epoch: 60, Training log loss: 0.178
Epoch: 80, Training log loss: 0.137
Epoch: 100, Training log loss: 0.116
Epoch: 120, Training log loss: 0.106
Epoch: 140, Training log loss: 0.096
Epoch: 160, Training log loss: 0.094
Epoch: 180, Training log loss: 0.089

```

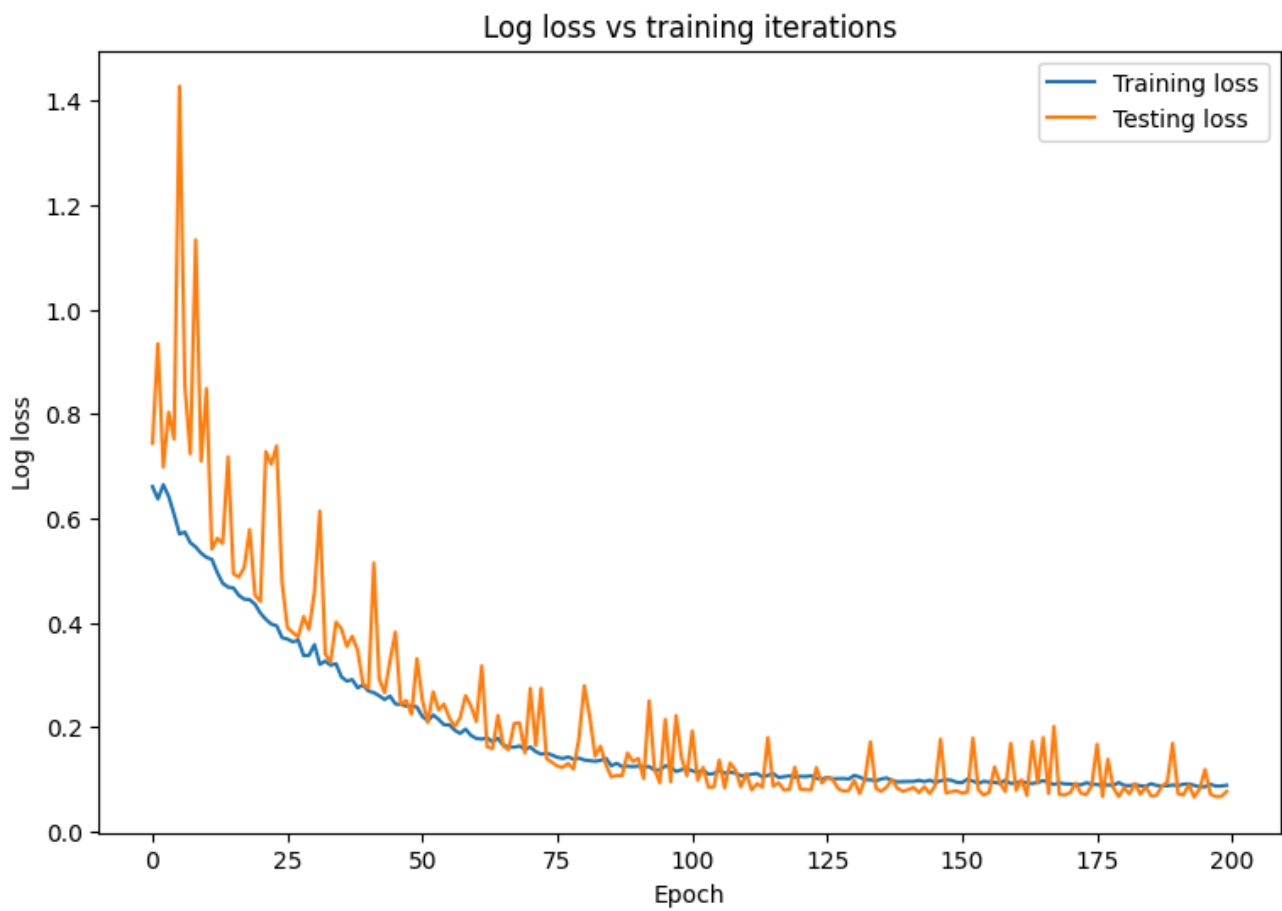
Performance evaluation

Observe the changes in your model's loss and accuracy over time.

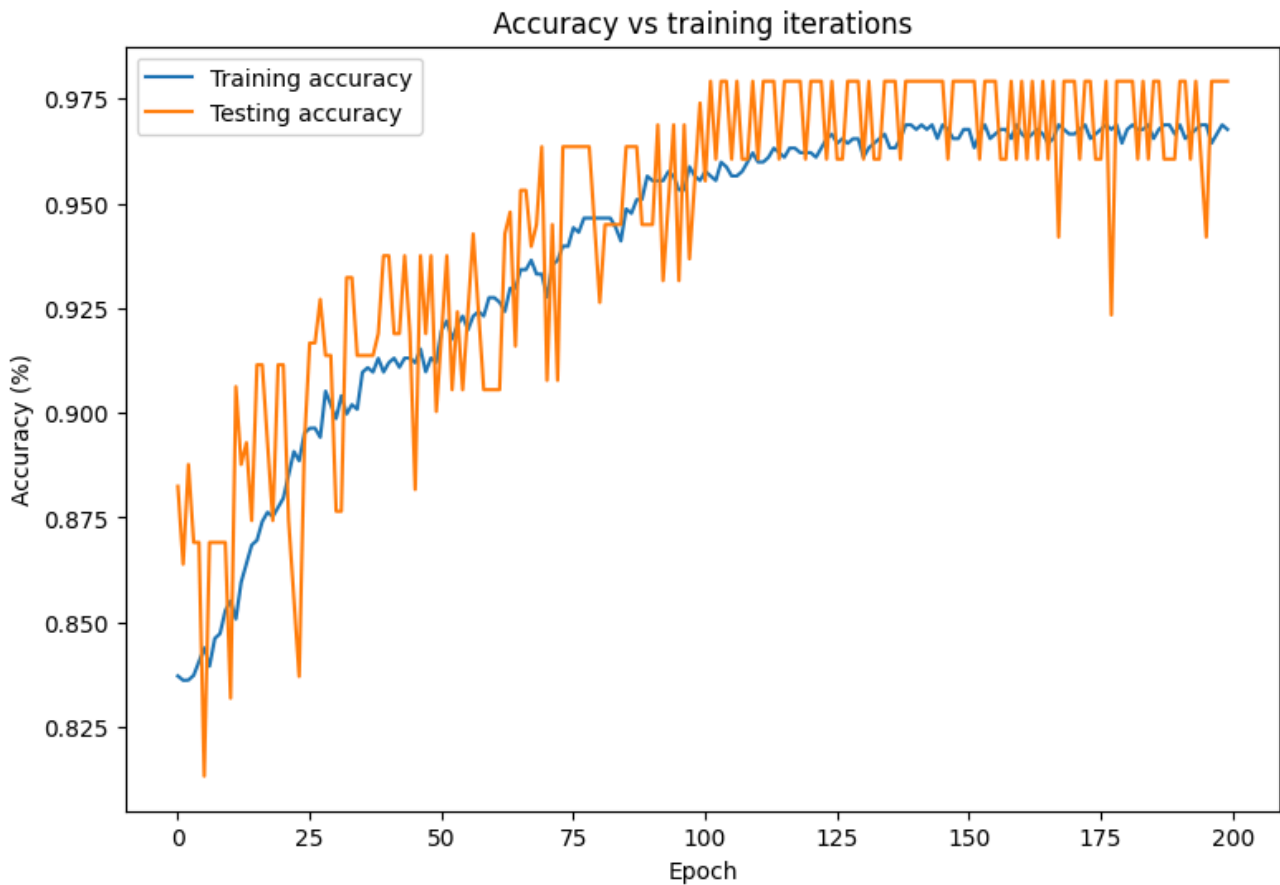
```

plt.plot(range(epochs), train_losses, label = "Training loss")
plt.plot(range(epochs), test_losses, label = "Testing loss")
plt.xlabel("Epoch")
plt.ylabel("Log loss")
plt.legend()
plt.title("Log loss vs training iterations");

```



```
plt.plot(range(epochs), train_accs, label = "Training accuracy")
plt.plot(range(epochs), test_accs, label = "Testing accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.legend()
plt.title("Accuracy vs training iterations");
```



```
print(f"Final training log loss: {train_losses[-1]:.3f}")  
print(f"Final testing log Loss: {test_losses[-1]:.3f}")
```

```
Final training log loss: 0.089  
Final testing log Loss: 0.077
```

```
print(f"Final training accuracy: {train_accs[-1]:.3f}")  
print(f"Final testing accuracy: {test_accs[-1]:.3f}")
```

```
Final training accuracy: 0.968  
Final testing accuracy: 0.979
```

The model demonstrates a high accuracy and a low loss when it comes to classifying tumors in the training dataset and also generalizes well to the unseen test data. To go one step further, you can explore error rates that give more insight beyond the overall accuracy score.

The two most popular error rates for binary classification problems are the false positive rate (FPR) and the false negative rate (FNR).

For this problem, the FPR is the proportion of malignant tumor predictions amongst tumors that are actually benign. Conversely, the FNR is the proportion of benign tumor predictions among tumors that are actually malignant.

Compute a confusion matrix using `sklearn.metrics.confusion_matrix`

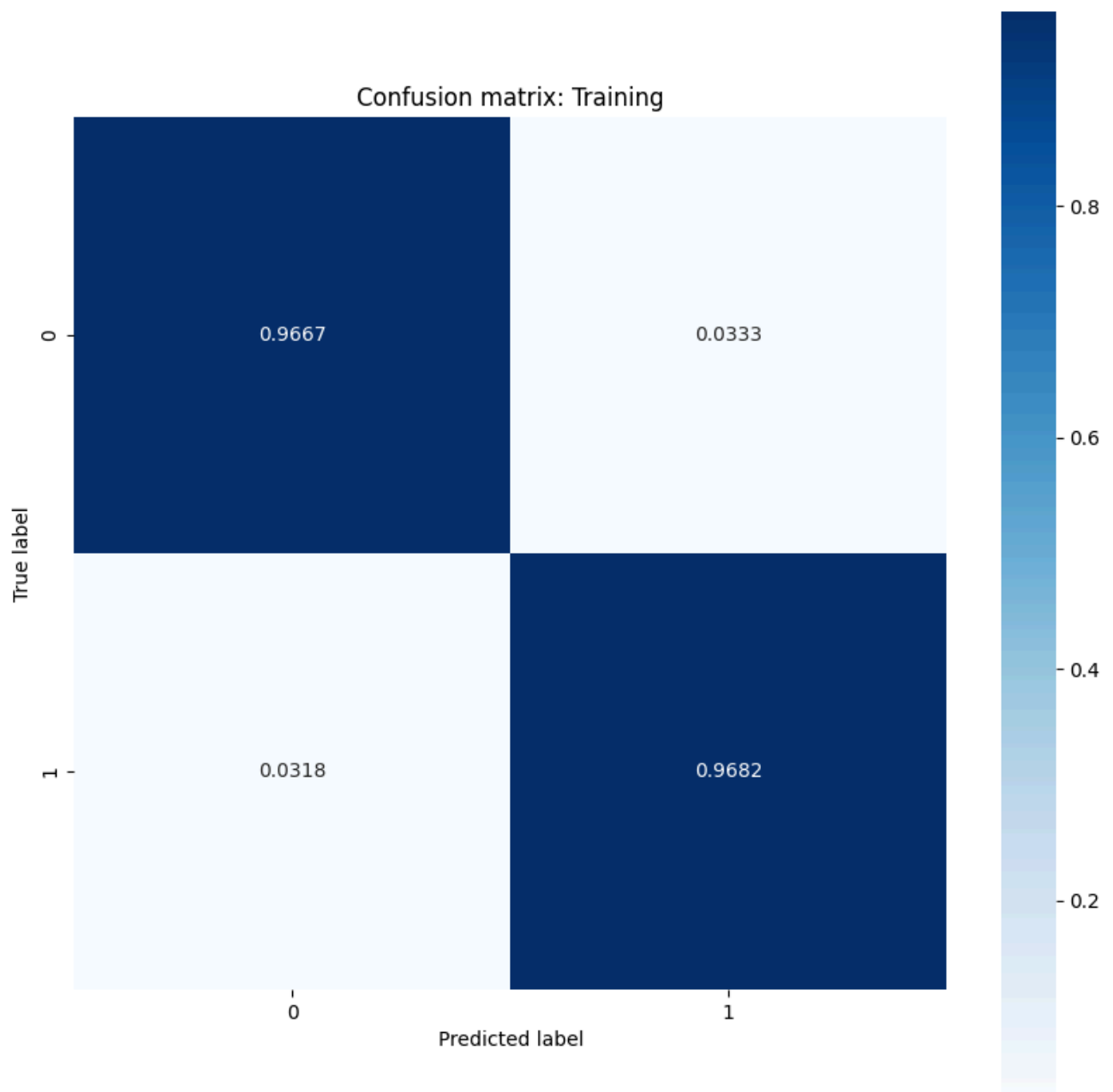
(https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html#sklearn.metrics.confusion_matrix)

, which evaluates the accuracy of the classification, and use matplotlib to display the matrix:

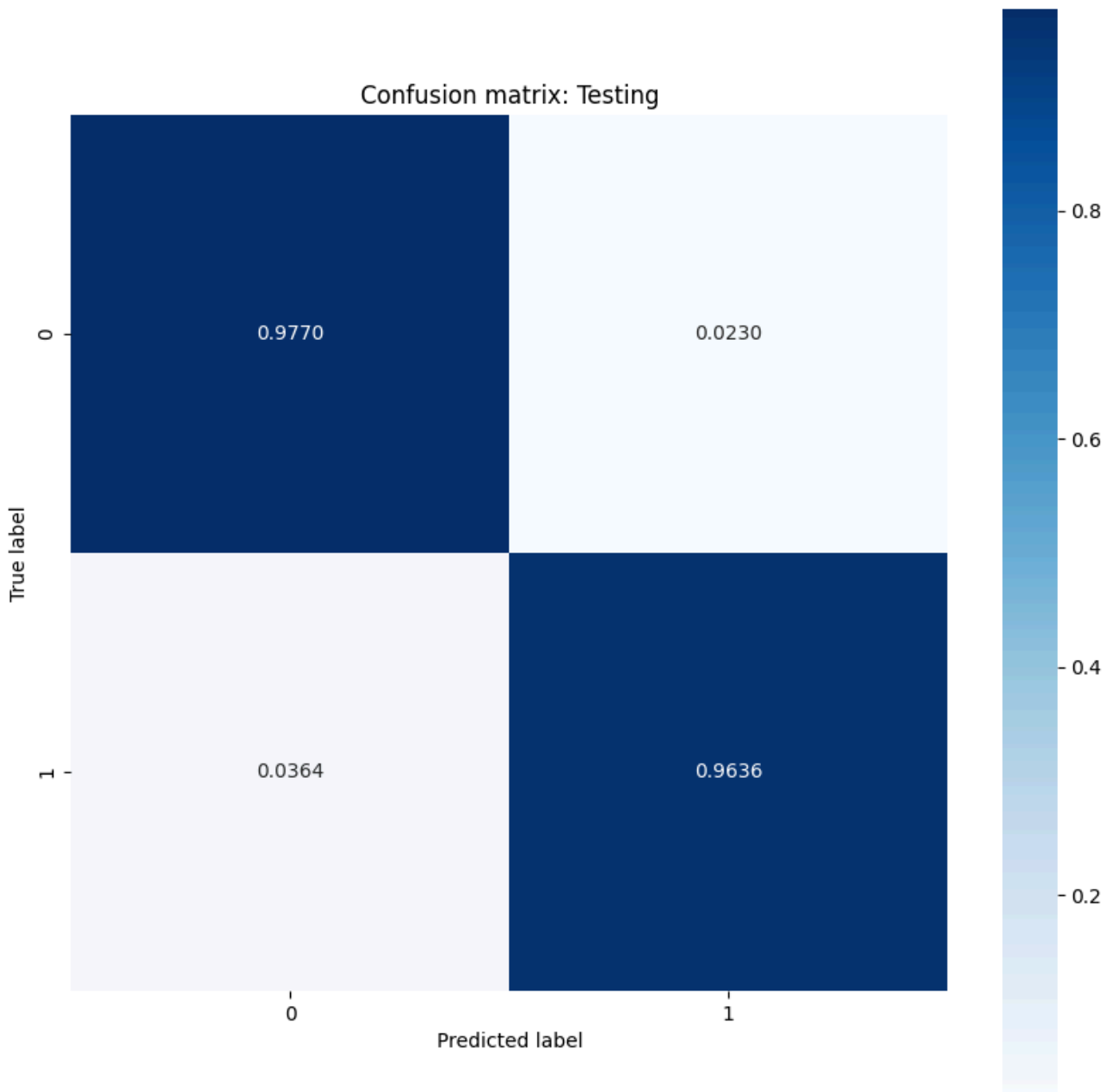
```
def show_confusion_matrix(y, y_classes, typ):
    # Compute the confusion matrix and normalize it
    plt.figure(figsize=(10,10))
    confusion = sklearn.metrics.confusion_matrix(y.numpy(), y_classes.numpy())
    confusion_normalized = confusion / confusion.sum(axis=1, keepdims=True)
    axis_labels = range(2)
    ax = sns.heatmap(
        confusion_normalized, xticklabels=axis_labels, yticklabels=axis_labels,
        cmap='Blues', annot=True, fmt='.4f', square=True)
    plt.title(f"Confusion matrix: {typ}")
    plt.ylabel("True label")
    plt.xlabel("Predicted label")

y_pred_train, y_pred_test = log_reg(x_train_norm, train=False), log_reg(x_test_norm, train=False)
train_classes, test_classes = predict_class(y_pred_train), predict_class(y_pred_test)

show_confusion_matrix(y_train, train_classes, 'Training')
```

```
show_confusion_matrix(y_test, test_classes, 'Testing')
```



Observe the error rate measurements and interpret their significance in the context of this example. In many medical testing studies such as cancer detection, having a high false positive rate to ensure a low false negative rate is perfectly acceptable and in fact encouraged since the risk of missing a malignant tumor diagnosis (false negative) is a lot worse than misclassifying a benign tumor as malignant (false positive).

In order to control for the FPR and FNR, try changing the threshold hyperparameter before classifying the probability predictions. A lower threshold increases the model's overall chances of making a malignant tumor classification. This inevitably increases the number of false positives and the FPR but it also helps to decrease the number of false negatives and the FNR.

Save the model

Start by making an export module that takes in raw data and performs the following operations:

- Normalization
- Probability prediction
- Class prediction

```
class ExportModule(tf.Module):
    def __init__(self, model, norm_x, class_pred):
        # Initialize pre- and post-processing functions
        self.model = model
        self.norm_x = norm_x
        self.class_pred = class_pred

    @tf.function(input_signature=[tf.TensorSpec(shape=[None, None], dtype=tf.float32)])
    def __call__(self, x):
        # Run the `ExportModule` for new data points
        x = self.norm_x.norm(x)
        y = self.model(x, train=False)
        y = self.class_pred(y)
        return y

log_reg_export = ExportModule(model=log_reg,
                              norm_x=norm_x,
                              class_pred=predict_class)
```

If you want to save the model at its current state, you can do so with the `tf.saved_model.save` (https://www.tensorflow.org/api_docs/python/tf/saved_model/save) function. To load a saved model and make predictions, use the `tf.saved_model.load` (https://www.tensorflow.org/api_docs/python/tf/saved_model/load) function.

```
models = tempfile.mkdtemp()
save_path = os.path.join(models, 'log_reg_export')
tf.saved_model.save(log_reg_export, save_path)
```

```
INFO:tensorflow:Assets written to: /tmpfs/tmp/tmp9k_sar52/log_reg_export/assets
INFO:tensorflow:Assets written to: /tmpfs/tmp/tmp9k_sar52/log_reg_export/assets
```

```
log_reg_loaded = tf.saved_model.load(save_path)
test_preds = log_reg_loaded(x_test)
test_preds[:10].numpy()
```

```
array([1., 1., 1., 1., 0., 1., 1., 1., 1., 1.], dtype=float32)
```

Conclusion

This notebook introduced a few techniques to handle a logistic regression problem. Here are a few more tips that may help:

- The [TensorFlow Core APIs](https://www.tensorflow.org/guide/core) (<https://www.tensorflow.org/guide/core>) can be used to build machine learning workflows with high levels of configurability
- Analyzing error rates is a great way to gain more insight about a classification model's performance beyond its overall accuracy score.
- Overfitting is another common problem for logistic regression models, though it wasn't a problem for this tutorial. Visit the [Overfit and underfit](https://www.tensorflow.org/tutorials/keras/overfit_and_underfit) (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit) tutorial for more help with this.

For more examples of using the TensorFlow Core APIs, check out the [guide](https://www.tensorflow.org/guide/core) (<https://www.tensorflow.org/guide/core>). If you want to learn more about loading and preparing data, see the tutorials on [image data loading](https://www.tensorflow.org/tutorials/load_data/images) (https://www.tensorflow.org/tutorials/load_data/images) or [CSV data loading](https://www.tensorflow.org/tutorials/load_data/csv) (https://www.tensorflow.org/tutorials/load_data/csv).

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 4.0 License](https://creativecommons.org/licenses/by/4.0/) (<https://creativecommons.org/licenses/by/4.0/>), and code samples are licensed under the [Apache 2.0 License](https://www.apache.org/licenses/LICENSE-2.0) (<https://www.apache.org/licenses/LICENSE-2.0>). For details, see the [Google Developers Site Policies](https://developers.google.com/site-policies) (<https://developers.google.com/site-policies>). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2024-08-15 UTC.

