Machine Overlord And You

Presented by Jon, Paris, Tim and Mars at OSCON 2018.

Description

Welcome to Machine Overlord and You! In this workshop, we'll build a collection of practical machine learning examples that make use of a variety of machine learning technologies present on Apple platforms.

Intro

Core ML is Apple's framework for working with trained ML models.

- It supports lots of different types of model, including:
- 30+ types of neural network layers
- Standard models like SVMs, generalized linear models, tree ensembles

Core ML makes it very straightforward for feeding data into a model and getting back results. It generates code for you that simplifies interaction with the model, making it easier to build something useful or cool.

Turi Create is a Python package that simplifies the process of creating machine learning models. It can also generate a Core ML model file for you to drop straight into your code.

The workflow for this workshop will be:

- We'll generate or download a model
- We'll integrate it into an app
- We'll test it out

With all that done, let's get started!

Setting Up

To get started, we'll download the project and data.

Workspace Setup

- Set up your workspace.

```
mkdir oscon_ai_2018
cd oscon_ai_2018

# Install turicreate
sudo pip install turicreate

# (If you're using pipenv, virtualenv, or some other Python setup that you
# prefer, use whatever you need to install 'turicreate'.)
```

NOTE

If you're using the Python that comes with macOS, you may need to tell pip to not try to upgrade 'six' (which you can't do, even as root, because of System Integrity Protection):

```
sudo pip install turicreate --upgrade --ignore-installed six
```

- Download and extract the empty source code.
- Download the data you need:

```
./setup.sh
```

- Open the Xcode project you downloaded.
- Open Preferences, and sign in to your Apple Developer account.

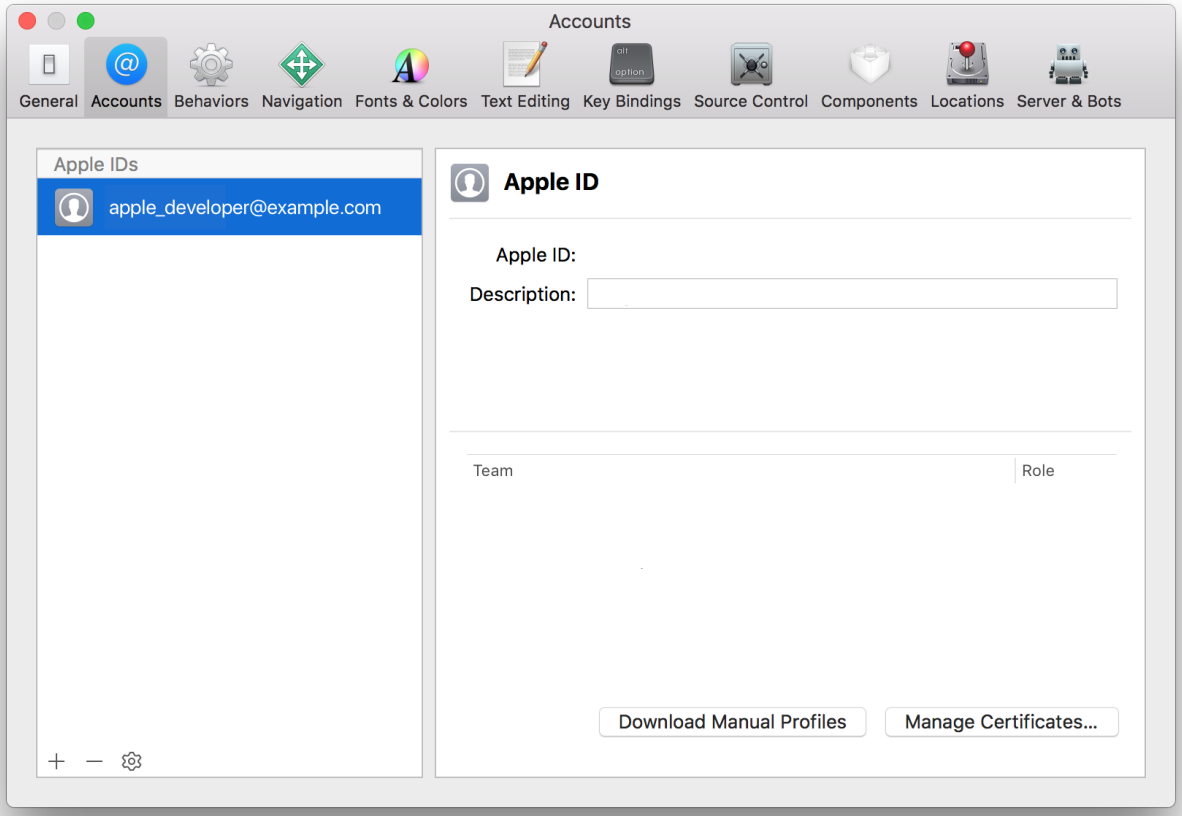

Figure 1. Signing into a developer account.

- Go to the Project settings, and set the bundle identifier and code signing team.

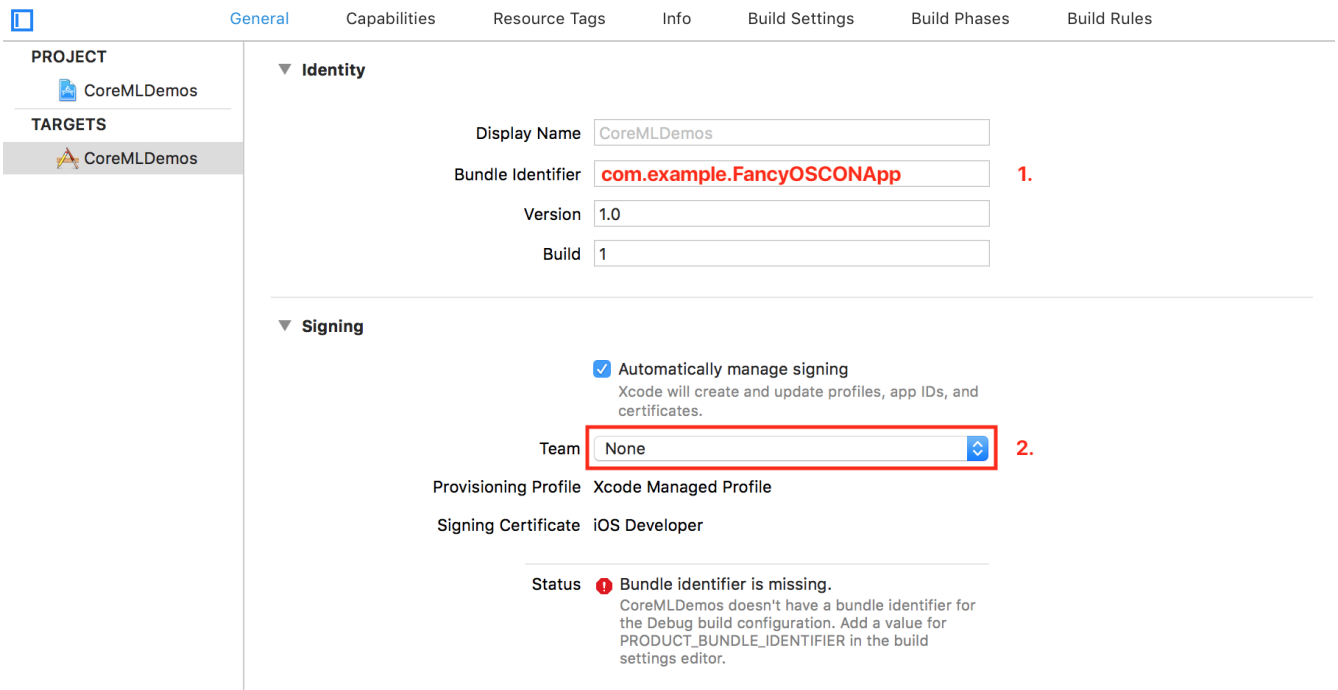

Figure 2. Configuring the app's bundle identifier and code signing team.

You're all set to go.

Downloading the data manually

If you prefer, you can download the data yourself.

Text Classification

First, review the license of the data.

- Download text classification data:

```
mkdir -p datasets/sentiment_sentences/
curl https://s3.amazonaws.com/amazon-reviews-
pds/tsv/amazon_reviews_us_Major_Appliances_v1_00.tsv.gz | gunzip >
datasets/sentiment_sentences/amazon_reviews.tsv
```

Handwritten Image Classification

```
pushd models
curl -O https://s3-us-west-2.amazonaws.com/coreml-models/MNIST.mlmodel
popd
```

Image Classification

- Download the Cats and Dogs image dataset
  - https://www.microsoft.com/en-us/download/details.aspx?id=54765
- Make the directory and unzip the data:

```
mkdir datasets/cats_dogs
pushd datasets/cats_dogs
unzip <PATH TO THE CATS AND DOGS ZIP FILE>
popd
pushd models
curl -O https://docs-assets.developer.apple.com/coreml/models/VGG16.mlmodel
popd models
```

Style Transfer

```
pushd models
# We rename these to remove the hyphens from the names, which causes problems in Xcode
curl https://s3-us-west-2.amazonaws.com/coreml-models/FNS-Udnie.mlmodel -o
FNSUdnie.mlmodel
curl https://s3-us-west-2.amazonaws.com/coreml-models/FNS-Candy.mlmodel -o
FNSCandy.mlmodel
curl https://s3-us-west-2.amazonaws.com/coreml-models/FNS-Feathers.mlmodel -o
FNSFeathers.mlmodel
curl https://s3-us-west-2.amazonaws.com/coreml-models/FNS-La-Muse.mlmodel -o
FNSLaMuse.mlmodel
curl https://s3-us-west-2.amazonaws.com/coreml-models/FNS-Mosaic.mlmodel -o
FNSMosaic.mlmodel
curl https://s3-us-west-2.amazonaws.com/coreml-models/FNS-The-Scream.mlmodel -o
FNSTheScream.mlmodel
popd
```

You're now ready to go.

Using a Sentiment Classifier Model

We'll now train a sentiment classifier model, and use it in an app.

A sentiment classifier model is one that can predict whether some text has a positive sentiment (happy) or negative sentiment (unhappy). We'll create the model ourselves, using Amazon review data.

Creating the Model

The data is product review data from Amazon, in gzipped TSV format; the file will be decompressed to your local computer.

Next, create the Core ML model by running this Python code:

```
#!/usr/bin/env python

import sys
import os
import turicreate as tc

# The location of the input data
DATA_LOCAL = "datasets/sentiment_sentences/amazon_reviews.tsv"

# Check that the file is there
if not os.path.exists(DATA_LOCAL):
    print("%s does not exist.", DATA_LOCAL)
    sys.exit(1)
```

```
# Read the data
reviews = tc.SFrame.read_csv(DATA_LOCAL, delimiter='\t', header=True)

# Select the specific columns we want
reviews = reviews['review_body', 'star_rating']

# Label each review based on star rating; >4 stars is positive, <4 stars is negative
reviews['sentimentClass'] = reviews['star_rating'].apply(lambda rating: 'positive' if
rating >= 4 else 'negative')

# Remove the star rating column; we don't need it anymore
reviews.remove_column('star_rating')

# Split the reviews into positive and negative
positive = reviews[reviews['sentimentClass'] == 'positive']
negative = reviews[reviews['sentimentClass'] == 'negative']

# We want an even number of positive and negative reviews, so pick the list
# that has the shorter amount...
review_count = min(len(positive), len(negative))

# And trim both lists to that count
positive = positive.head(review_count)
negative = negative.head(review_count)

# Now combine them back together
reviews = positive.append(negative)

# Save the SFrame for later use
MODEL_PATH = "amazon_reviews.sframe"
reviews.save(MODEL_PATH)

# Create the model! We're telling it to look at the 'review_body' column as its input,
# and the 'sentimentClass' column as the label.
model = tc.sentence_classifier.create(reviews, 'sentimentClass', features=
['review_body'])

# Evaluate this model
evaluation = model.evaluate(reviews)

# Print the evaluation
print(evaluation)

# Export the model into a form that Core ML can use
COREML_MODEL_PATH = "SentimentClassifier.mlmodel"
model.export_coreml(COREML_MODEL_PATH)

print("Created model at {}".format(COREML_MODEL_PATH))
```

This trains a model using this data, and generates a Core ML model that classifies bags of words as

positive or negative.

Using it in Code

We'll now add code that makes use of the created `.mlmodel` file.

- Add the .mlmodel file to the project.
- Go to SentimentAnalysisViewController.swift
- Add the following variables

```
// The CoreML model we'll use to perform the classifications
let sentimentModel = SentimentClassifier()

// A tagger that can break up text into a series of words
let tagger = NSLinguisticTagger(tagSchemes: [.tokenType], options: 0)
```

- Add the following method

```
// Given a string, returns a dictionary containing the word count for each
// unique word.
func bagOfWords(from text: String) -> [String: Double] {

    // The dictionary we'll send back
    var result : [String: Double] = [:]

    // NSLinguisticTagger hasn't been updated to use Swift's range types,
    // so we use the older NSRange type.

    // Create an NSRange that refers to the entire length of the input.
    let range = NSRange(location: 0, length: text.utf16.count)

    // Create an option set that indicates to the tagger that we want
    // to skip all punctuation and whitespace.
    let options: NSLinguisticTagger.Options = [.omitPunctuation, .omitWhitespace]

    // Provide the text to the tagger.
    tagger.string = text

    // Loop over every token in the sentence.
    tagger.enumerateTags(in: range, unit: .word, scheme: .tokenType, options: options)
{
        _, tokenRange, _ in

        // This block will be called for each token (i.e. word) in the
        // text.

        // Get the region of the input string that contains this token
        let word = (text as NSString).substring(with: tokenRange)

        // Increment the number of times we've seen this word.
        result[word, default: 0] += 1
    }

    // Return the summed word counts.
    return result
}
```

- Add this code to the `sentiment(for text:)` method:

```
// Get the bag of words from the text
let bagOfWords = self.bagOfWords(from: text)

if bagOfWords.count == 0 {
    // No words. Nothing to classify.
    return (nil, nil)
}

do {
    // Perform the prediction using this bag of words
    let prediction = try sentimentModel.prediction(text: bagOfWords)

    // Get the predicted class
    let sentimentClass = prediction.sentimentClass

    // Get the probability of the predicted class
    let sentimentProbability = prediction.sentimentClassProbability[sentimentClass] ??
0

    // Indicate this sentiment
    return (Sentiment(rawValue: prediction.sentimentClass), sentimentProbability)

} catch {
    return (nil, nil)
}
```

- Test the app by running it.

NOTE	iOS 12 and macOS 10.14 include NLTokenizer, which is a more modern API, and sidesteps this for you - NSLanguageTokenizer works on older versions, and also lets you see a bit more of the process

Detecting Handwritten Digits

MNIST is a dataset containing a large number of pictures of handwritten digits. We'll create an app that lets the user draw a number, and then try to recognise what they drew.

- Add the MNIST classification model to the project.
- Go to `DigitRecognitionViewController`
- Add the following code to `performRecognition`:

```
// Get the image that we're going to analyse
let originalImage = scribbleView.captureImage()

// The model expects an image that's 28x28, so we need to resize our
// image to this size. (This is also why the ScribbleView is square - it
// means that the image will be the correct shape that the model is
// expecting.)
let size = CGSize(width: 28, height: 28)

guard let resizedImage = originalImage.resize(to: size) else {
    fatalError("Failed to resize image")
}

// We need to convert the image into a pixel buffer of the correct format.
// because that's the type of data that the model is expecting.
guard let pixelBuffer = resizedImage.pixelBuffer() else {
    fatalError("Failed to resize and create pixelbuffer")
}

// Create an instance of the model and make a prediction
guard let result = try? MNIST().prediction(image: pixelBuffer) else {
    fatalError("Failed to create prediction")
}

// Get the class label that we matched on.
let detectedNumber = result.classLabel

// Display the number.
resultLabel.text = String(detectedNumber)
```

- Run the app. When you draw into the view, it will try to figure out which digit you drew.

The other way you can do this is use the Vision framework to process the image for you.

- Add the following method:

```
func handleClassificationResult(_ request: VNRequest, _ error: Error?) {
    guard let result = request.results?.first as? VNClassificationObservation else {
        return
    }

    DispatchQueue.main.async {
        self.resultLabel.text = result.identifier
    }
}
```

- Add the following variables:

```
lazy var model = VNCoreMLModel(for: MNIST().model)

lazy var request = VNCoreMLRequest(model: model, completionHandler:
handleClassificationResult)
```

- Add the following code to `performRecognitionWithVision`:

```
// Get the image we're about to analyse
let originalImage = scribbleView.captureImage()

// Create a handler that processes this specific image.
let handler = VNImageRequestHandler(cgImage: originalImage.cgImage!, options: [:])

// Run the handler through the request. Its completion handler will
// execute after analysis is complete, which will set the label's text.
try? handler.perform([request])
```

- Adjust the code in `touchesEnded` to call `performRecognitionWithVision`.

Training and Using an Image Classifier Model with the Camera

An image classifier predicts the contents of an image, based on a pre-trained model.

Creating the Model

We'll use the cats and dogs dataset we downloaded earlier.

- Run this code:

```
#!/usr/bin/env python

import turicreate as tc

DATA_PATH = "datasets/cats_dogs/PetImages"

print("Loading data...")

# Load the images into an SFrame; also include a column that contains the path
# Not all images are valid, but that's fine, since we have so many of them
data = tc.image_analysis.load_images(DATA_PATH, with_path=True)

# Create a label column from the path
data['label'] = data['path'].apply(lambda path: 'dog' if '/Dog' in path else 'cat')


COUNT_PER_CLASS=50

print("Limiting to {} images per class".format(COUNT_PER_CLASS))

cats = data[data['label'] == 'cat'].head(COUNT_PER_CLASS)
dogs = data[data['label'] == 'dog'].head(COUNT_PER_CLASS)

data = cats.append(dogs)

print("Creating model...")

# Create the model - it will automatically detect the image column, but we must
provide
# the column that contains the labels
model = tc.image_classifier.create(data, target='label')

# Save the trained model for later use in Turi Create, if we want it
model.save("CatDogClassifier.model")

# Export the model for use in Core ML
model.export_coreml('CatDogClassifier.mlmodel')
```

- This will load and train the model. It took about 30 minutes to train on my MacBook Pro (2.7 GHz Intel Core i7, NVIDIA GeForce GT 650M.) When it's done, a Core ML model called CatsAndDogs will be produced.

NOTE	Because this takes a while, we'll move on to Using a Style Transfer Model and come back later.

Using the model in iOS

- Go to ImageClassificationViewController.swift

- Add the following method:

```
// Called when the VNCoreMLRequest has finished classifying.
func handleClassificationResult(_ request: VNRequest, _ error: Error?) {

    // The type of the results depends on the model, so we don't know at
    // build time what they'll be. In this case, because the model is
    // a classifier, the results will be of type VNClassificationObservation,
    // so we'll cast to that (or bail out if that fails)
    guard let results = request.results as? [VNClassificationObservation] else {
        return
    }

    // Get up to four results from the classifier
    let firstResults = results.prefix(upTo: min(4, results.count-1))

    // Build a list of strings that combine the predictions with their
    // probabilities (expressed as a percentage)
    var resultStrings : [String] = []

    for result in firstResults {
        let id = result.identifier
        let confidence = Int(result.confidence * 100)
        resultStrings.append("\(id) (\(confidence)%)")
    }

    // We can only update the view from the main queue
    DispatchQueue.main.async {

        // Update the label
        self.resultLabel.text = resultStrings.joined(separator: "\n")

        // Indicate that we want to wait timeBetweenClassifications until
        // the next classification
        self.nextClassification = Date(timeIntervalSinceNow:
self.timeBetweenClassifications)
    }
}
```

- Add the following variables:

```
// The model that the CoreML Request will use
lazy var model = VNCoreMLModel(for: CatDogClassifier().model)

// A request that uses the model, and calls handleClassificationResult when
// it's done
lazy var request = VNCoreMLRequest(model: model, completionHandler:
handleClassificationResult)
```

- Add the following code to `handle(pixelBuffer:)`:

```
// If the next classification date is in the future, do nothing with
// this frame
if Date() < nextClassification {
    return
}

// Create a handler that uses this pixel buffer
let handler = VNImageRequestHandler(cvPixelBuffer: pixelBuffer)

// Run our request through the handler
try! handler.perform([request])
```

- Test the app. It will report on what it sees.
- Try using a different model, like `VGG16`, which classifies a much broader range of objects.

Using a Style Transfer Model

- We won't train our own here because it takes forever (like 2 days on my laptop); instead, we'll download some pre-trained ones.
- Add the `FNS` models to the project
- Go to `StyleTransferViewController`
- Add the following method:

```
func handleImageProcessingResult(_ request: VNRequest, _ error: Error?) {
    // Ensure that we got a VNPixelBufferObservation to use
    guard let result = request.results?.first as? VNPixelBufferObservation else {
        return
    }

    // Get the pixel buffer from the result
    let pixelBuffer = result.pixelBuffer

    // Create a CIImage from this pixel buffer
    let ciImage = CIImage(cvPixelBuffer: pixelBuffer)

    // Create a UIImage that uses this CIImage, and specify its scale
    // and orientation
    let image = UIImage(ciImage: ciImage, scale: CGFloat(1.0), orientation:
UIImageOrientation.left)

    DispatchQueue.main.async {
        // Update the image view
        self.resultImageView.image = image

        // Signal that we're done processing this
        self.processing = false
    }
}
```

- Add the following method:

```
// Produces a request, given a CoreML model.
func request(for model: MLModel) -> VNCoreMLRequest {

    // Create a VNCoreMLModel that wraps this MLModel
    let model = try! VNCoreMLModel(for: model)

    // Produce the request
    return VNCoreMLRequest(
        model: model,
        completionHandler: handleImageProcessingResult
    )
}
```

- Add the following property:

```
// The initial request uses the FNSTheScream model
lazy var request : VNCoreMLRequest = self.request(for: FNSTheScream().model)
```

- Update the `@IBAction` methods to include the following code:

```
// Each of these methods replaces the current model with a different one

@IBAction func selectTheScream(_ sender: Any) {
    request = request(for: FNSTheScream().model)
}

@IBAction func selectFeathers(_ sender: Any) {
    request = request(for: FNSFeathers().model)
}

@IBAction func selectCandy(_ sender: Any) {
    request = request(for: FNSCandy().model)
}

@IBAction func selectLaMuse(_ sender: Any) {
    request = request(for: FNSLaMuse().model)
}

@IBAction func selectMosaic(_ sender: Any) {
    request = request(for: FNSMosaic().model)
}

@IBAction func selectUdnie(_ sender: Any) {
    request = request(for: FNSUdnie().model)
}
```

- Add the following code to the awakeFromNib method:

```
// Tell the session to run this code when a new frame arrives off the
// camera
session.frameDelivered = {

    // Are we in the middle of processing a frame?
    if self.processing {
        // Do nothing with it
        return
    }

    // Flag that we're busy
    self.processing = true

    // Run the pixel buffer through the model
    let handler = VNImageRequestHandler(cvPixelBuffer: $0)
    try! handler.perform([self.request])
}
```

- Add the following code to `saveImage`:

```
guard let image = resultImageView.image?.ciImage else {
    return
}

// We need to transfer the image data from the GPU to the CPU. It's
// currently in a CIImage; we need to convert it to a CGImage, which
// can be saved.
let context = CIContext(options: nil)

// Get a CGImage from the CIImage.
guard let cgImage = context.createCGImage(image, from: image.extent) else {
    return
}

// Construct a UIImage that refers to the CGImage.
let imageWithCGImage = UIImage(cgImage: cgImage)

// Save this UIImage to the photo library.
PHPhotoLibrary.shared().performChanges({
    PHAssetCreationRequest.creationRequestForAsset(from: imageWithCGImage)
})

resultImageView.alpha = 0

UIView.animate(withDuration: 0.25) {
    self.resultImageView.alpha = 1
}
```

- Test the app. It will apply the selected style to what the camera sees.

NOTE	We're now probably ready to move back to Training and Using an Image Classifier Model with the Camera.

Detecting Faces

The `Vision` framework has built-in tools for detecting faces in images. Using it, you can find both the areas of an image where a face is likely to be, as well as the positions of *facial landmarks*, which are things like the nose, lips, eyebrows, and so on.

- Go to `FaceDetectionViewController`
- Add the following method:

```
func handleRequestResults(_ request: VNRequest, _ error: Error?) {

    guard let observations = request.results as? [VNFaceObservation] else {
        // No observations
        return
    }

    DispatchQueue.main.async {

        // Construct paths for both face outlines and features
        let outlinesPath = CGMutablePath()
        let landmarksPath = CGMutablePath()

        let size = self.cameraView.bounds.size

        let flipped = self.cameraView.position == .front

        // For each face, add a box around it.
        for face in observations {

            // Get a path that draws a rectangle around the face
            let faceBox = self.boundingBoxPath(for: face, in: size, flipped: flipped)

            // Add it to the path that contains face outlines
            outlinesPath.addPath(faceBox)

            // Get a path that draws each individual feature of the face
            let landmarks = self.landmarksPath(for: face, in: size, flipped: flipped)

            // Add it to the path that contains face features
            landmarksPath.addPath(landmarks)
        }

        // Update the paths we're showing
        self.faceLayer.path = outlinesPath
        self.faceLandmarksLayer.path = landmarksPath
    }
}
```

- Add the following property:

```
// Create a request to detect faces.
lazy var request = VNDetectFaceLandmarksRequest(completionHandler:
handleRequestResults)
```

- Add the following code to the `handle(pixelBuffer:)` method:

```
// We've received a pixel buffer from the camera view. Use it to
// ask Vision to detect faces.
let handler = VNImageRequestHandler(cvPixelBuffer: pixelBuffer)

do {
    try handler.perform([request])
} catch let error {
    print("Error perfoming request: \(error)")
}
```

- Add the following code to `boundingBoxPath(for:, in:, flipped:)`

```
// The bounding box is normalized - (0,0) is bottom-left, (1,1) is top-right

// We want to rotate it so that (0,0) is top-left, so we'll flip it
// on the Y axis, and if we need to flip it horizontally, we'll do the
// same thing on the X axis; this pushes it off-screen, so we'll push it back
// by adding 1 to both axes. We'll then scale it to the size of the camera
// view.

let rect = face
    .boundingBox
    .applying(CGAffineTransform(scaleX: flipped ? -1 : 1, y: -1)) // flip it
    .applying(CGAffineTransform(translationX: flipped ? 1 : 0, y: 1)) // move it back
    .applying(CGAffineTransform(scaleX: size.width, y: size.height))

path.addRect(rect)
```

- Add the following code to `landmarksPath(for:, in:, flipped:)`

```
// We'll need to flip two things: first, the bounding box of the face,
// and second, the features themselves. Create a flip transform and store
// it.
let flipTransform =
    CGAffineTransform(scaleX: flipped ? -1 : 1, y: -1)
        .concatenating(CGAffineTransform(translationX: flipped ? 1 : 0, y: 1))

// Flip the bounding box.
let rect = face.boundingBox.applying(flipTransform)

// Convert it into the coordinates for the camera view.
let faceBounds = VNImageRectForNormalizedRect(rect, Int(size.width), Int(size.height))

if let landmarks = face.landmarks {
    // Landmarks are relative to, and normalized within, face bounds
    let affineTransform =
        CGAffineTransform(translationX: faceBounds.origin.x, y: faceBounds.origin.y)
        .scaledBy(x: faceBounds.size.width, y: faceBounds.size.height)
```

```
    let featureTransform = flipTransform.concatenating(affineTransform)

    // Treat eyebrows and lines as open-ended regions when drawing paths.
    let openLandmarkRegions: [VNFaceLandmarkRegion2D] = [
        landmarks.leftEyebrow,
        landmarks.rightEyebrow,
        landmarks.faceContour,
        landmarks.noseCrest,
        landmarks.medianLine
        ].compactMap({$0})

    for openLandmarkRegion in openLandmarkRegions{
        self.addPoints(in: openLandmarkRegion,
                       to: faceLandmarksPath,
                       applying: featureTransform,
                       closingWhenComplete: false)
    }

    // Draw eyes, lips, and nose as closed regions.
    let closedLandmarkRegions: [VNFaceLandmarkRegion2D] = [
        landmarks.leftEye,
        landmarks.rightEye,
        landmarks.outerLips,
        landmarks.innerLips,
        landmarks.nose
    ].compactMap({$0})

    for closedLandmarkRegion in closedLandmarkRegions {
        self.addPoints(in: closedLandmarkRegion,
                       to: faceLandmarksPath,
                       applying: featureTransform,
                       closingWhenComplete: true)
    }
}
```