HACKING WITH SWIFT

ADVANCED iOS

VOLUME TWO

COMPLETE TUTORIAL COURSE

Learn ARKit, Core ML, drag and drop, and m
with real-world pr ts

FREE SAMPLE

Paul Hudson

Project 1
Trade My Tesla

Ease yourself into machine learning with custom models.

Setting up

In this first project we're going to build an app to calculate the trade-in value of a Tesla car – that might seem simple, but it's intentional. You see, most of the app itself is just regular UIKit code because I want to focus on one of the most powerful, flexible, and *complicated* parts of iOS11: Core ML.

"ML" is short for "machine learning", and it's probably the single biggest computing buzzword at the time I write this. It's a whole other area of computing, with its own specialist jargon, its own algorithms, its own community, and its own learning curve. With iOS 11, Apple wrapped up a good chunk of ML functionality into a new framework called Core ML, and in doing so made it surprisingly easy for anyone to build machine learning into their app.

You might not realize it, but you use machine learning every day: when Netflix recommends movies to you, when iOS's Quick Type keyword suggests words to type, or when Facebook suggests pages you might like, that's all done using ML. It's everywhere, and now with Core ML it can be in your apps too.

We're going to be looking at Core ML more later in the book, but for now I want to give you a simple, useful introduction that equips you to write other kinds of ML apps easily.

Let's begin now: launch Xcode and make a new iOS project using the Single View App template. Name it "Project1", then make sure all three of the checkboxes are unchecked – we don't need Core Data, unit tests, or UI tests in this project or any others in this book.

Once the project is open, please go to the project settings and tell it to use only Portrait for its device orientation. You can make other orientations work if you want, but it's just a distraction here – there's nothing new or special involved.

I've included a simple logo for this app in the project1-files folder that came with this book's source code, so please copy the logo files into your asset catalog. And with that, it's time for some Python. Yes, you read that right: Python.

Machine learning 101

I don't want to rant on about how complex ML can be, or go into its history. Instead, I want to dive right in and make something useful, because Apple have really made it surprisingly accessible.

However, before I dive in there is a small amount of information you need to know first, so please bear with me – this stuff really is important, and I've tried to keep it as short as possible.

First, a definition of machine learning: it lets computers interpret new data based on previous data they have been exposed to.

Consider for a moment the complexity of trying to write an algorithm that detects whether a photo contains a guitar – you might try to look for guitar-shaped things, but of course guitars can be at all sorts of angles, plus they come in different colors and different shapes, or have different lighting.

ML lets us give a computer 1000 pictures of guitars to help it understand the variety of possibilities. Rather than describing our rules programmatically we're describing them *empirically* – we just keep throwing more data at the computer until it has a firm grasp of what a guitar looks like. Once the computer is ready, you should be able to send images of new guitars – guitars the computer has never seen before – and have them recognized.

This process of preparing a computer to recognize something is called *training*: you collect your input data and pass it through a learning algorithm until the computer has enough data to make predictions on new data. The end result is something called a trained model, which contains the computer's understanding of all its data – it doesn't contain all those images of guitars, but instead contains lots of rules the computer will use to try to identify new guitars.

Once you have a model, you can start to use it to make predictions about new data. This involves looking at the trained model to find correlations with new data that gets presented, then returning a result – it effectively says "other pictures that look similar to this were of guitars, so this one is probably a guitar too."

The learning algorithms and trained model predictions sound very difficult, and you won't be surprised to know they *are* very difficult, but here's the magic: we just don't care. You see, the learning algorithms have already been created by a variety of researchers around the world and are designed to accept a wide variety of data, and the trained model predictions are what Core ML provides – all *we* need to do is provide some training data, e.g. pictures of guitars, or house prices in Dallas, or weather forecasts for the last year, or stock market fluctuations in Japan, and so on.

That might sound relatively easy, but trust me: it isn't. In fact, the main reason that ML falls down is because training data can be hard to find. Sure, if you're Netflix you have hundreds of gigabytes of movie watching to draw on, but if you're just an independent developer it can be tricky to get sufficient data to allow Core ML to make accurate predictions.

Obviously it's down to you to find training data for your apps, because I can't predict how you'll use Core ML. However, I *am* going to show you how to take training data and turn it into a trained Core ML model, so you have all the skills you need to make your own models for whatever kind of ML app you can dream, up. Once we have the model, you'll see how easy it is to use in an app – Core ML really makes it easy!

Tip: If you want to experiment more with Core ML but don't have much data, visit **http://archive.ics.uci.edu/ml/** for a whole mine of data sets.

OK, enough theory: it's time for code. However, as I mentioned already we're going to be using *Python* and not *Swift*. That's not a typo: Apple uses Python to create its Core ML models, because many major machine learning tools are also written in Python. Don't worry: there isn't much Python code, and I'm going to break it down into small chunks so you can see exactly how it works.

All of Apple's Core ML Python tools are wrapped up in a Python package called "coremltools", but we also need to install two other tools: Scikit-learn and Pandas. Scikit-learn is a machine learning library that can do classification of text and images, grouping of similar objects, predicting of future values based on past data, and more. Pandas is a data analysis library that makes it easy to load and shape data - it's a bit like a spreadsheet.

Your Mac comes with Python pre-installed, but it won't come with those three modules or their dependencies. Fortunately, Python makes it easy to install all three using the command line. Open your Mac's Terminal app now, and run this command:

```
pip install -U coremltools scikit-learn pandas
```

Tip: If that command fails on your Mac it means you don't have the "pip" program installed. In that situation, run the command **sudo easy_install pip** first, then repeat the **pip** command.

That will install all three modules and their dependencies in one shot, giving us everything we need. Some readers have reported that adding "scipy" is required for them to follow along, but only install that if you find you're having problems.

Now, even though my Mac is new, its data isn't – I migrated it from a previous Mac, which in turn was migrated from a previous Mac, and so on. As a result, I've got a fairly convoluted and crufty system in places, and that simple command above threw up some errors.

If the same happens to you, try the below command instead – it uses "sudo" to gain admin privileges, and tries to re-install any packages rather than just upgrade:

```
sudo pip install --ignore-installed coremltools scikit-learn
pandas
```

That works for me even on my crufty system, so hopefully it will work for you too.

Once the packages are installed I'd like you to create a new folder on your desktop called "Cars", then copy cars.csv from my project files into that directory. This contains the training data we'll be using for this project, and I'd like you to open it in a text editor just quickly.

Here's a sample of what you'll see:

```
model,premium,mileage,condition,price
1,0,116235,2,55926
1,1,63917,2,57113
```

```
2,0,148908,2,68009
0,0,60578,1,16770
2,1,161468,1,35438
0,1,182157,0,7530
1,0,148226,3,43872
```

Let me explain what's going on:

1. This is a CSV, for "comma-separated values". That means it stores structured data in rows and columns, with each column separated by a comma.
2. The first line contains our field names: the model of the car, whether it has premium upgrades installed, the number of miles traveled, what condition its in, and what price it sold for.
3. Every column contains numbers. Sometimes these are self-explanatory, e.g. a mileage of 116235 means the car had traveled 116,235 miles. However, others – e.g. model – have to map to something else, in this case 0 is a Model 3, 1 is a Model S, and 2 is a Model X. This is important: Scikit-learn deals with floating-point numbers, not strings or other data types.
4. I've given you 1000 data points for training data, which ought to be good enough for Core ML to generate good predictions.

I've already explained that the "model" field can be one of three numbers, but there are two other fields that need to be mapped. The first is "premium", which will be 1 if the car has premium upgrades installed or 0 otherwise, and the second is "condition", which maps as follows: 0 is poor, 1 is OK, 2 is good, and 3 is great.

In order to use this data in iOS, we need to load it with Pandas, process it using Scikit-learn, then save it as a Core ML model. But before we do *that* I need to write one big warning:

Warning: This training data was created by me for this project, and has little to no basis on real-world sales prices of Tesla – I'm not responsible if a dealer offers you a wildly different sum of money for your car, and I'm not associated with Tesla in any way!

OK: let's write some Python to process our training data. Create a new text file called

convert_cars.py in the same Cars directory as cars.csv, then open it for editing in a text editor. You can use Xcode if you want, but there are alternatives such as Sublime Text or Atom that do a great job too.

The first three lines of code we need to write are to bring in the Scikit-learn, Pandas and CoreMLTools modules. We're going to bring in all of Pandas and CoreMLTools, but from Scikit-learn we only need to bring in its linear regression class, which is a very well-known statistical algorithm that takes some numerical input values and attempts to find between those values and some result. In our case, the result is the trade-in value of a car, and the other values are its model, condition, mileage, and so on.

So, add these three lines of code to the top of your Python file:

```
from sklearn.linear_model import LinearRegression
import pandas
import coremltools
```

Next we need to ask Pandas to load the CSV, and here's where you'll see why Pandas is so useful – it takes just one line of code to load and parse our CSV file. Add this line of code below the previous two lines:

```
data = pandas.read_csv("cars.csv")
```

If you want to see the effect of that, you can try printing it out by putting this line below:

```
print(data)
```

You can save the script and run it using "python convert_cars.py" – you should see the data from our CSV all neatly formatted, thanks to Pandas working its magic.

The next two lines are a bit confusing at first, but they do the critical work: we need to ask Scikit-learn to do the work of analyzing all our numbers and draw a virtual line in an attempt to find correlations. That alone is straightforward, but the syntax is quite confusing at first glance because Pandas does curious things with brackets.

First, let me show you the code – please add these two lines below the previous code:

```
model = LinearRegression()
model.fit(data[["model", "premium", "mileage", "condition"]],
data["price"])
```

Now let me explain what's happening, because the double brackets after **data** seem quite bizarre at first. The reason it's strange is because Pandas overrides one of those two sets of brackets, so they don't mean the same thing.

The inner brackets – that's the **["model", "premium", "mileage", "condition"]** part – means just what it means in Swift: this is a list of things, which in our case is the list of fields we want Scikit-learn to evaluate. The outer brackets are Pandas's indexing operator, which is how you read data.

If we had written **data["model"]** it would mean "read precisely one field," but we want to read a *list* of values, so we pass that list in. So, putting the two together this means "read this list of values from the data that Pandas parsed," and that gives us the input for our model: lots of cars with a variety of models, conditions, mileages, and so on.

As for the output, that comes right at the end: **data["price"]**. As a result, we're asking Scikit-learn to look at the values for model, premium, mileage, and condition, and try to figure out how they influence the price value. The syntax is a bit messy in Python, but I hope you can at least agree that it's a huge amount of work being done in just two lines of code!

At this point we have a fully trained machine learning model, but we can't use it in Core ML just yet. You see, Core ML has its own custom model data format that gets optimized by Xcode during the build process so that it can be efficient on mobile devices, so we need to convert our Scikit-learn model to a Core ML model.

This takes just one line of code using CoreMLTools: you can ask it to convert a model, passing in a list of the names of its inputs and a name of its output. We already have the data fields configured, so all this does is provide *names* to be used in the finished Core ML model, which

in turn surface in our Swift code.

Add this line of code now:

```
coreml_model = coremltools.converters.sklearn.convert(model,
["model", "premium", "mileage", "condition"], "price")
```

The next step is optional, but recommended: we can add a little metadata to our model, describing who made it, what license it is released under, and what it's supposed to do. These will appear in Xcode, so it's a good idea to fill them in.

Add these three lines:

```
coreml_model.author = "Hacking with Swift"
coreml_model.license = "CC0"
coreml_model.short_description = "Predicts the trade-in price
of a Tesla car."
```

Finally, all we need to do is save the Core ML model so we can use it in iOS. So, add this final line of code to your Python script:

```
coreml_model.save("Cars.mlmodel")
```

Note the capital C in the filename: this gets used as your class name in Swift, so calling it Cars.mlmodel will ensure we get a class called **Cars**.

That's it: no more Python now. You should be able to run "python convert_cars.py" from the command line to have the CSV successfully converted into Cars.mlmodel, which is a trained Core ML model ready to predict new trade-in car prices based on data it hasn't seen before.

Note: Although the training data is quite large, the resulting Core ML model is small – don't worry!

In the future, I'm hopeful Apple will expand Core ML to allow model creation directly on iOS,

because that would allow us to train models dynamically based on new user data rather than using data we trained ourselves. It would also mean I wouldn't need to dedicate the first project chapter of my book to teaching you some Python, but for now it's all we have.

We're done with Python, CoreMLTools, and the Terminal app now, so please drag Cars.mlmodel into your Xcode project navigator – let's get back to Xcode!

Note: Sometimes Xcode causes issues when you drag in Cars.mlmodel – it would often delete the file rather than add it, and even when it added it correctly it wouldn't add it to the current target, which means it wouldn't be built with the project. If you don't see the usual modal window asking you how you want to copy in the file, I suggest you do the following:

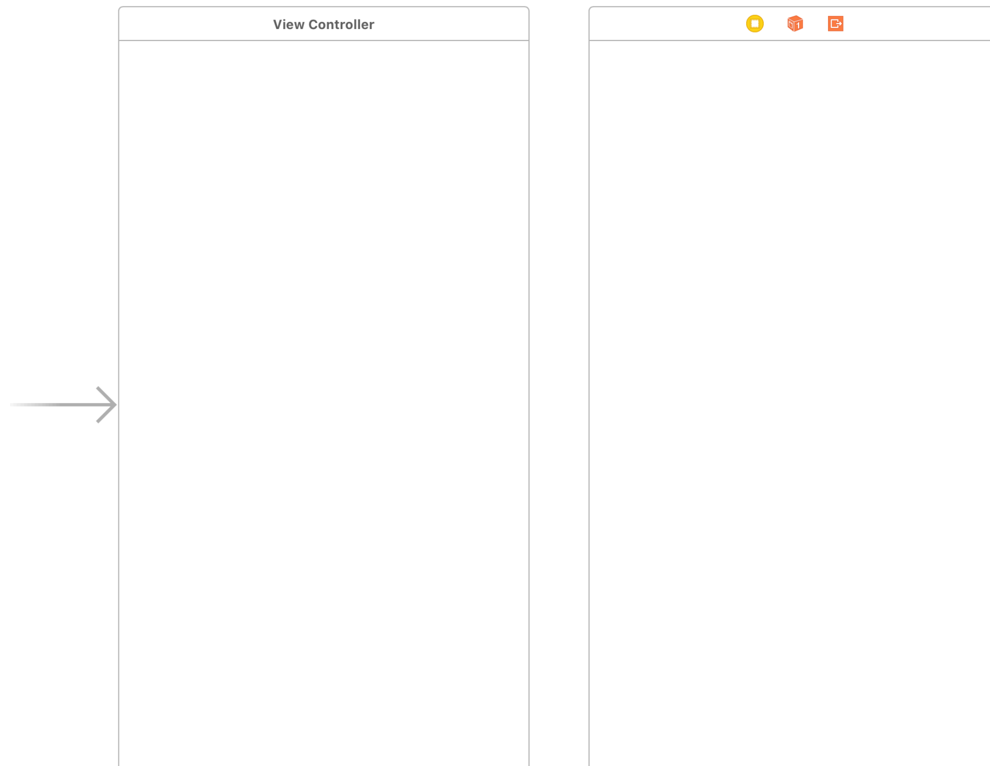
1. Use Finder to copy the file from the "Cars" directory into your project's directory.
2. Drag it into the project navigator.
3. Select Cars.mlmodel, then go to the file inspector (Alt+Cmd+1) and check the box marked "Project1" under Target Membership.

Named colors and stack view spacing

The user interface for this project isn't complex, so I'm going to spice it up just a little and show you two small new features in iOS 11 along the way.

Open Main.storyboard in Xcode, and you'll see you have one empty view controller like usual. We're going to use that screen to display a logo and disclaimer, then add a second view controller to display the main controls for our app.

First, drag out a new view controller to the right of the existing one. Use the identity inspector to remove the class of the left-hand controller so that it's a default **UIViewController**, then make the right-hand controller have the **ViewController** class. It's a small thing, but it means we don't need to create a second view controller class for our controls screen – we'll use the one Xcode gave us instead, because we don't need anything for the left-hand controller.



Of course, we don't really need the left-hand controller at all: it will just show a logo and a disclaimer, but as this is a test app the disclaimer is hardly necessary. However, there is method to my madness: we're using two view controllers because I want to demonstrate a new feature in iOS 11 called *named colors*.

It's common to have a specific set of colors used in your app: a main color that fits your logo color, a secondary color that complements the main color, and perhaps a few other shades or tints for the rest of your user interface.

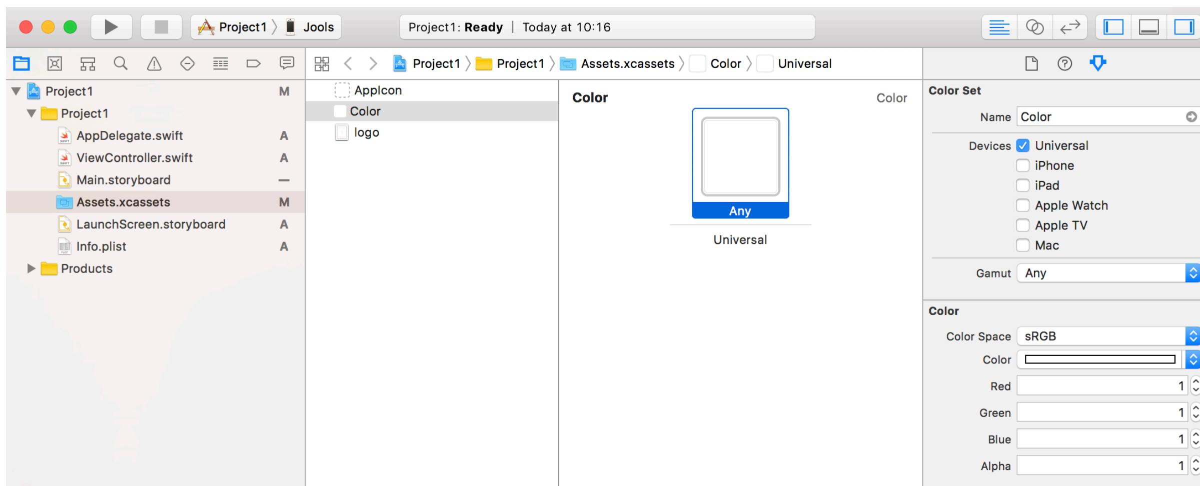
Most developers accomplished by declaring **UIColor** instances in code, which solved the problem but not terribly well – designers can't make quick changes to try things out, you can't see the colors in IB, and you also need to store color values as ranges from 0.0 to 1.0 rather than the more traditional RGB ranges as 0 to 255.

iOS 11 solves this problem by allowing us to create and name colors in asset catalogs, then load them using a new **UIColor** initializer called **UIColor(named:)**. These colors are also

available inside IB, so all three problems are solved: designers can make a change in one place to affect the entire app, the colors can be used in IB as well as code, and you also get to create them using a regular macOS color well rather than floating-point numbers.

Named color are created and managed inside an asset catalog, so please select Assets.xcassets to open it for editing. You'll see "AppIcon" and "logo" are already in there, but I'd like you to right-click in the empty whitespace below them and choose "New Color Set" – that should insert a new color called "Color", and select it for editing.

The controls to edit a color are all stored in the attributes inspector, which you can activate by pressing Alt+Cmd+4. You'll see it has a name ("Color" right now) and a list of devices you want to use it on. If you select the empty color well in the center pane – where it says "Any" and "Universal" – you should see more options appear in the attributes inspector, letting you select your color.



I'd like you to change the color's name from "Color" to "TeslaRed". You can use spaces in the name if you want, but given that most other names *don't* have spaces it's nice to be consistent.

For the color value, change Color Space to Display P3, then enter the following values:

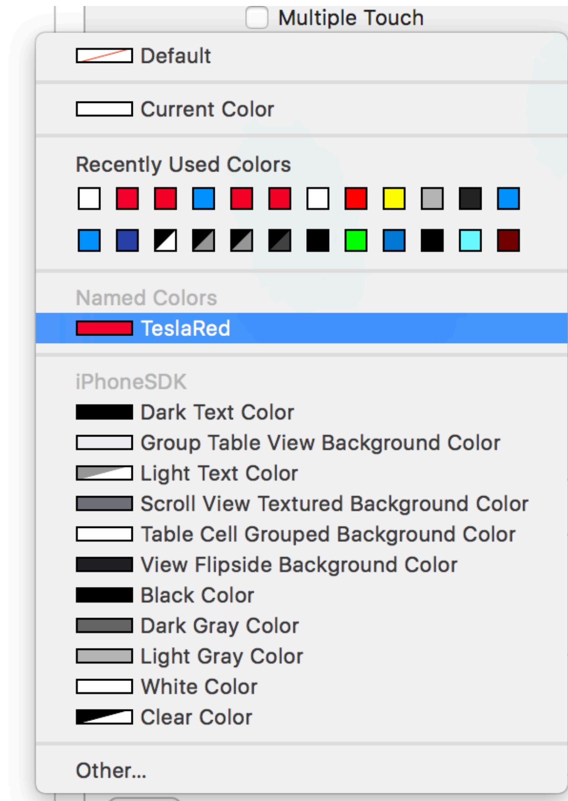
- Red: 0.882
- Green: 0.0967
- Blue: 0.216

- Alpha: 1

That should give you some thing extremely close to Tesla's crimson red color. And that's it: we now have a color we can use both in code and in Interface Builder. Even better, we can use it in lots of places, and have the whole app update when it gets changed.

Head back to Main.storyboard, because it's time to use our new named color as the background for both of our view controllers. Click inside the first controller on the IB canvas to activate its view, then look in the attributes inspector for the Background option. I need you to click the *arrows* next to the Background option, not the color itself – clicking the color activates Apple's color selector, but clicking the arrows activates a dropdown menu where you can select predefined colors.

Inside that colors dropdown you should see "TeslaRed" under the heading Named Colors – that's the color we just created. If Xcode is playing ball, you should be able to select that and see a red color on your canvas; if you're hitting the same bug many other users are, you'll just see black. You should also use "TeslaRed" as the background color of the second view controller too, so they both match.



Now that we have a nice Tesla color, let's drop in the (totally not approved by their marketing department) "Trade My Tesla" logo and a couple of other controls. You already added the logo images to your asset catalog, so we just need to add it to IB. We're also going to add a short piece of disclaimer text, and a button to make the main part of the app launch.

So, please add these user interface controls:

- A **UIImageView** at X:21, Y:40, width: 333, height: 422. Give it the image "logo".
- A **UILabel** at X:45, Y:560, width: 285, height: 41. Make it a white color, change its number of lines to 2, then center its text. Finally, give it the text "This app is not endorsed by Tesla, and any price is entirely fictional."
- A **UIButton** at X:127, Y:603, width: 120, height: 44. Give it the title "PROCEED" in white, then make it use the font System Heavy 25.

We need to add some Auto Layout rules so that our design looks good on other devices, but Xcode isn't able to guess them directly for us so we need to do them by hand for a change.

It's not too hard, but I suggest you use the document outline to make your life easier. Add these constraints:

- From "logo" to "View" add Leading Space to Safe Area, Trailing Space to Area, Center Horizontally in Area, then Top Space to Safe Area.
- From "PROCEED" to "PROCEED" add Height, then from "PROCEED" to "View" add Center Horizontally in Safe Area and Bottom Space to Safe Area.
- From the disclaimer label to the disclaimer label add Width, then from the disclaimer label to "PROCEED" add Vertical Spacing and Center Horizontally.

Now, some of the Auto Layout rules for the logo might seem redundant: why specify leading and trailing edge as well as centering it horizontally? To get an idea why – and also to see there's more work still to do – I'd like you to look for "View as: iPhone 7" at the bottom of

your IB canvas. This tells us we're currently previewing sizes as if this were an iPhone 7 device, but if you click that you'll see a selection of other devices – try clicking two to the right, which activates the iPhone SE size.

What you'll see is that the Tesla logo gets stretched because Auto Layout tries to preserve its natural height alongside our constraints, and it's also far too close to the text at the bottom.

This is easily fixed: return to "View as: iPhone 7" then create a constraint from the logo to itself and choose "Aspect Ratio – this forces the image to resize itself vertically in order to match the decreased horizontal space, and it now works correctly when previewed with iPhone SE.

Note: The curious among you will note that it still doesn't work with the iPhone 4s size, but that's OK – that device doesn't support iOS 11, or even iOS 10 for that matter.

That's our first view controller designed, so now it's on to the second. For this we're going to use a stack view, partly because it makes layout easy, but partly so I can show you another helpfully little feature introduced in iOS 11 – sorry, I can't help myself!

Drag a vertical stack view into the second view controller. We need to add to it quite a few components, and I find it *significantly* less error-prone to drag them into the stack view using the document outline.

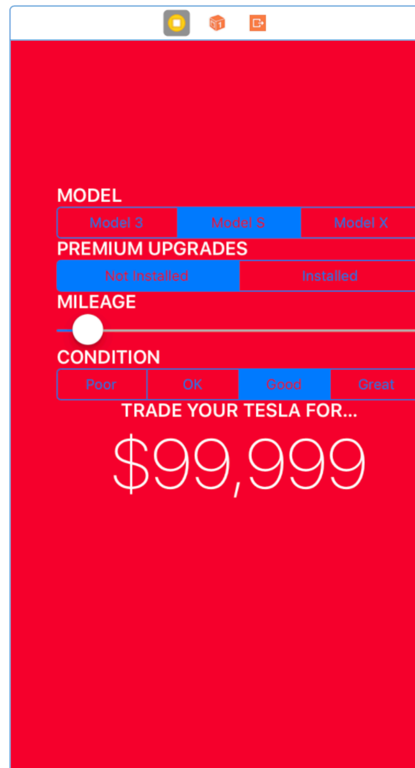
Add these components to the stack view now:

- A **UILabel** with the text "MODEL". Give it the color white and the font style "Headline".
- A segmented control with three segments: Model 3, Model S, and Model X. Make Model S selected.
- A **UILabel** with the text "PREMIUM UPGRADES". Give it the color white and the font

style “Headline”.

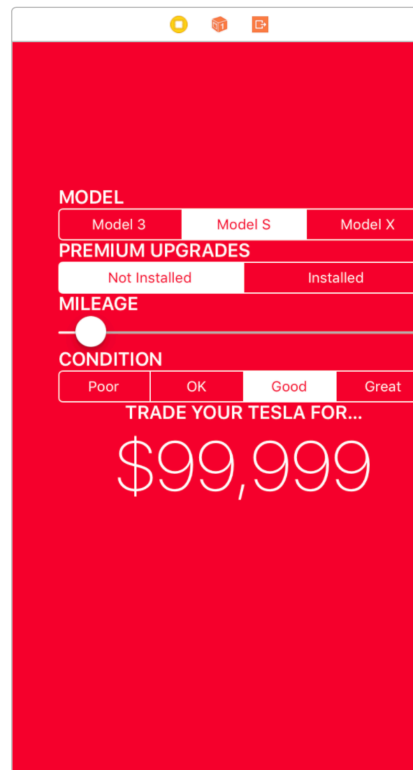
- A segmented control with two segments: Not Installed and Installed.
- A third **UILabel** with the text “MILEAGE”, and the same color and font style as the others.
- A slider with value 10,000, minimum 0, and maximum 200,000.
- A fourth **UILabel** with the text “CONDITION”, and the same color and font style as the others.
- A segmented control with four segments: Poor, OK, Good, and Great. Make Good selected.
- A fifth **UILabel** with the text “TRADE YOUR TESLA FOR…”, and the same color and font style as the others. Make this one aligned center, though.
- One last **UILabel** with the text “$99,999”. Give the color white, but change its font to be System Thin 64 and its alignment to be center.

They should all automatically stack up vertically inside the stack view, and when you’re done it should look something like this:



It looks pretty awful right now, but we can make it look a little better with two changes.

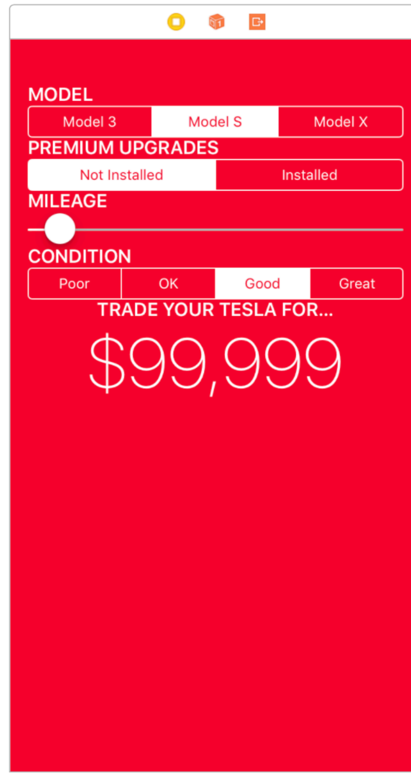
First, we need to get those segmented controls using a white tint. You could tint them all individually if you want, but it's easier to select the main view (the red area) and change *its* tint to white. You should get this:



Second, we need the stack view to take up all available space horizontally, then align itself to the top. This can be done using three Auto Layout constraints: Ctrl-drag from the stack view to its parent view, then hold down Shift and choose Leading Space to Safe Area, Trailing Space to Safe Area, and Top Space to Safe Area before clicking Add Constraints.

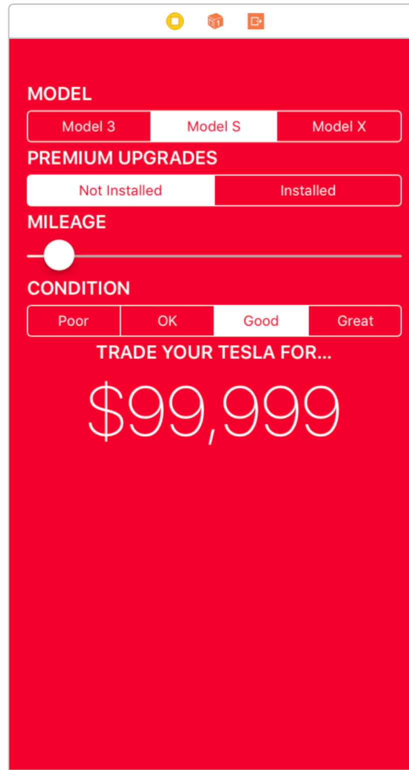
That will create constraints based on the stack view's current position, so we need to tweak them a little. Open the size inspector (Alt+Cmd+5, then use the Edit button for each constraint to make them all have the Constant value 20.

With that change things should look even better, although still far from perfect:



To make the layout perfect, we need to add a little spacing to our stack view. Stack views do have a spacing attribute right there to edit in the attributes inspector, but it's used to set *consistent* item spacing for all items in the stack view – it can't be used to specify individual spacing for items. We'll still use it, though: select the stack view, then set its Spacing attribute to 5. As for the rest of the spacing, we'll need to do that in code.

Here's how it should look so far:



Now I'd like you to switch to the assistant editor and make a variety of outlets:

- Connect the stack view to an outlet called **stackView**.
- Connect the model segmented control to an outlet called **model**.
- Connect the premium upgrades segmented control to an outlet called **upgrades**.
- Connect the "MILEAGE" label to an outlet called **mileageLabel**. We'll be using that to show whatever value is selected in the slider.
- Connect the mileage slider to an outlet called **mileage**.
- Connect the condition segmented control to an outlet called **condition**.
- Connect the large $99,999 label to an outlet called **valuation**.

We also need to connect actions to the three segmented controls and the slider, but I'd like you to connect them all to the *same* action. So, create an action for the segmented control called **calculateValue()**, then connect the other three controls to that same action by Ctrl-dragging directly onto the **calculateValue()** method stub. This means the same method will be called no