**8.Building Machine Learning Detectors:Makine Öğrenimi Dedektörlerini Yapma**

Kİtabın sonunda oluşturulan tool nasıl kullanacağımı anlatmış bir bak

Bugün, makine öğrenimi sistemlerini uygulamanın ağır matematiksel kaldırmasını işleyen yüksek kaliteli açık kaynaklı yazılım sayesinde, temel Python'u bilen ve temel kavramları anlayan herkes makine öğrenimini kullanabilir.

Bu bölümde machine larning malware detection sistemleri için kullanılan scifit-learn ,en ünlü ve en iyi olan kendi düşüncesine göre,mevcut açık kaynak makine öğrenme paketini gösterecek.Bu bölüm birçok örnek kod içerir. Ana kod bloklarına malware\_data\_science / ch8 / code dizininden erişilebilir ve ilgili örnek verilere, bu kitaba eşlik eden kod ve verilerdeki (ve sanal makinedeki) malware\_data\_science / ch8 / data dizininden erişilebilir.Metni takip ederek, örnek kodu inceleyerek ve verilen örnekleri deneyerek, bölümün sonunda kendi makine öğrenimi sistemlerinizi oluştururken ve değerlendirirken rahat olmalısınız.Ayrıca, genel bir kötü amaçlı yazılım algılayıcı oluşturmayı ve belirli kötü amaçlı yazılım aileleri için kötü amaçlı yazılım algılayıcıları oluşturmak için gerekli araçları kullanmayı da öğrenirsiniz.Burada edindiğiniz beceriler, kötü amaçlı e-postaları veya şüpheli ağ akışlarını tespit etme gibi diğer güvenlik sorunlarına makine öğrenimi uygulamanıza olanak tanıyan geniş bir uygulamaya sahip olacaktır.

     İlk olarak, scikit-learn'ü kullanmadan önce bilmeniz gereken terminoloji ve kavramları öğrenirsiniz. Ardından, Bölüm 6'da öğrendiğiniz karar ağacı kavramlarına dayalı temel bir karar ağacı algılayıcısı uygulamak için scikit learn'ü kullanırsınız.Daha sonra, gerçek dünyadaki özellik çıkarma ve kötü amaçlı yazılımları tespit etmek için rastgele bir orman yaklaşımı kullanan gerçek bir kötü amaçlı yazılım algılayıcısı oluşturmak için özellik çıkarma kodunu scikitlearn ile nasıl entegre edeceğinizi öğreneceksiniz.Son olarak, örnek rastgele orman dedektörü ile makine öğrenimi sistemlerini değerlendirmek için scikit-learn'ün nasıl kullanılacağını öğrenirsiniz.

->Terminology and Concepts:

Önce bazı terminolojiyi gözden geçirelim. Açık kaynak kitaplık scikit-learn (kısaca sklearn), hem güçlü hem de kullanımı kolay olduğu için makine öğrenimi topluluğunda popüler hale geldi. Birçok veri bilimcisi, kitaplığı bilgisayar güvenliği topluluğu içinde ve diğer alanlarda kullanır ve çoğu, makine öğrenimi görevlerini gerçekleştirmek için ana araç kutusu olarak ona güvenir. Sklearn, yeni makine öğrenimi yaklaşımlarıyla sürekli olarak güncelleniyor olsa da, bu makine öğrenimi yaklaşımlarının kullanımını basitleştiren tutarlı bir programlama arayüzü sağlar.

Birçok makine öğrenimi çerçevesi gibi sklearn de vektör biçiminde eğitim verileri gerektirir. Vektörler, dizideki her dizinin eğitim örnek yazılım ikili programının tek bir özelliğine karşılık geldiği sayı dizileridir. Örneğin, makine öğrenimi algılayıcımızın kullandığı yazılım ikili dosyalarının iki özelliği sıkıştırılmışsa ve şifrelenmiş veriler içeriyorsa, eğitim örneği ikili dosyası için özellik vektörümüz [0,1] olabilir. Burada, vektördeki ilk dizin, ikilinin sıkıştırılıp sıkıştırılmadığını temsil eder, sıfır "hayır" ı gösterir ve ikinci dizin, ikili dosyanın şifrelenmiş veri içerip içermediğini gösterir ve biri "evet" i gösterir.

Her bir dizinin hangi özelliğe eşlendiğini hatırlamanız gerektiğinden, vektörlerle çalışmak garip olabilir. Neyse ki sklearn, diğer veri temsillerini vektör biçimine çeviren yardımcı kod sağlar. Örneğin, eğitim verilerinizin sözlük temsillerini (örneğin, {"sıkıştırılmış": 1, "şifrelenmiş veriler içerir": 0}) sklearn'ın üzerinde çalıştığı vektör gösterimine dönüştürmek için sklearn'ın DictVectorizer sınıfını kullanabilirsiniz, örneğin [0 , 1]. Daha sonra, vektörün indisleri ile orijinal özellik adları arasındaki eşlemeyi kurtarmak için DictVectorizer'ı kullanabilirsiniz.

Sklearn tabanlı bir detektörü eğitmek için, sklearn için iki ayrı nesneyi aktarmanız gerekir: özellik vektörleri feature vectors(daha önce açıklandığı gibi) ve bir etiket vektörü a label vector.Bir etiket vektörü, eğitim örneği başına bir sayı içerir ve bizim durumumuzda, örneğin kötü amaçlı yazılım veya zararlı yazılım olup olmadığına karşılık gelir.Örneğin, sklearn için üç eğitim örneğini ve ardından etiket vektörünü [0,1,0] geçirirsek, sklearn'a ilk örneğin iyi huylu yazılım, ikinci örneğin kötü amaçlı yazılım ve üçüncüsünün iyi huylu yazılım olduğunu söyleriz.Geleneksel olarak, makine öğrenimi mühendisleri eğitim verilerini temsil etmek için bir büyük harf X değişkeni ve etiketleri temsil etmek için küçük bir y değişkeni kullanır.Durumdaki fark, matrisleri temsil eden değişkenlerin büyük harfle yazılması (vektör dizileri olarak düşünebiliriz) ve ayrı vektörleri temsil eden küçük harfli değişkenlerin matematiğindeki geleneği yansıtır. Bu kuralın çevrimiçi makine öğrenimi örnek kodunda kullanıldığını göreceksiniz ve bu kuralı bu kitabın geri kalanında rahat etmeniz için kullanıyorum.

Sklearn çerçevesi, sizin de yeni bulabileceğiniz diğer terminolojiyi kullanır. Makine öğrenimi tabanlı dedektörlere "dedektörler detectors" demek yerine, sklearn bunları "sınıflandırıcılar classifiers" olarak adlandırıyor. Bu bağlamda, sınıflandırıcı terimi basitçe, nesneleri iki veya daha fazla kategoriye ayıran bir makine öğrenimi sistemi anlamına gelir. Bu nedenle, bir algılayıcı (bu kitapta kullandığım terim), nesneleri kötü amaçlı yazılım ve zararlı yazılım malware and benignware gibi iki kategoriye ayıran özel bir sınıflandırıcı türüdür. Ayrıca, eğitim terimini kullanmak yerine sklearn dokümantasyonu ve API genellikle fit terimini kullanır. Örneğin, "eğitim örneklerini kullanarak bir makine öğrenimi sınıflandırıcısını uydur fit a machine learning classifier using training examples" gibi bir cümle görürsünüz. Bu, "eğitim örneklerini kullanarak bir makine öğrenimi algılayıcısını eğitin rain a machine learning detector using training examples" demeye eşdeğerdir.

Son olarak, sınıflandırıcılar bağlamında algılama terimini kullanmak yerine, sklearn tahmin terimini kullanır. Bu terim, sklearn çerçevesinde ve daha genel olarak makine öğrenimi topluluğunda, bir görevi gerçekleştirmek için bir makine öğrenimi sistemi kullanıldığında, bir hafta sonra bir hisse senedinin değerini tahmin etmek veya bilinmeyen bir ikili dosyanın kötü amaçlı yazılım olup olmadığını tespit etmek için kullanılır.

->Building a Toy Decision Tree–Based Detector:Oyuncak Karar Ağacı Tabanlı Dedektör Oluşturma

Artık sklearn’in teknik terminolojisine aşina olduğunuza göre, sklearn çerçevesini kullanarak 6. Bölümde tartıştığımız satırlar boyunca basit bir karar ağacı oluşturalım. Karar ağaçlarının, bu vektörlerin kötü niyetli mi yoksa zararsız mı olduğuna ilişkin bir karara varmak için girdi vektörleri hakkında bir dizi soru sordukları “20 soruluk” bir oyun türü oynadıklarını hatırlayın. Adım adım bir karar ağacı sınıflandırıcısı oluşturup ardından eksiksiz bir program örneğini keşfediyoruz. 8-1 listesi, gerekli modüllerin sklearn'dan nasıl içe aktarılacağını gösterir.

from sklearn import tree

from sklearn.feature\_extraction import DictVectorizer

Listing 8-1: Importing sklearn modules

İçe aktardığımız ilk modül olan ağaç, sklearn’in karar ağacı modülüdür. İkinci modül olan feature\_extraction, sklearn'ın DictVectorizer sınıfını içe aktardığımız yardımcı modülüdür. DictVectorizer sınıfı, okunabilir, sözlük biçiminde sağlanan eğitim verilerini, makine öğrenimi dedektörlerini gerçekten eğitmek için sklearn'ın gerektirdiği vektör gösterimine uygun bir şekilde çevirir.

Sklearn'den ihtiyacımız olan modülleri içe aktardıktan sonra, Liste 8-2'de gösterildiği gibi gerekli sklearn sınıflarını başlatıyoruz.

classifier = (1)tree.DecisionTreeClassifier()

vectorizer = (2)DictVectorizer(sparse=(3)False)

Listing 8-2: Initializing the decision tree classifier and vectorizer

Oluşturduğumuz ilk sınıf olan DecisionTreeClassifier (1), dedektörümüzü temsil eder. Sklearn, karar ağacımızın tam olarak nasıl çalışacağını kontrol eden bir dizi parametre sağlasa da, burada sklearn’in varsayılan karar ağacı ayarlarını kullanmak için herhangi bir parametre seçmiyoruz.

Örneklediğimiz bir sonraki sınıf DictVectorizer (2). Yapıcıda False (3) 'e seyrek olarak ayarladık ve sklearn'a hafızadan tasarruf eden ancak çalışmak için karmaşık olan seyrek vektörleri kullanmasını istemediğimizi söylüyoruz. Sklearn’in karar ağacı modülü seyrek vektörleri kullanamadığından, bu özelliği kapatıyoruz.

Artık sınıflarımızı başlattığımıza göre, Liste 8-3'te gösterildiği gibi bazı örnek eğitim verilerini başlatabiliriz.

# declare toy training data

(1)training\_examples = [

{'packed':1,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':0},

{'packed':1,'contains\_encrypted':1},

{'packed':1,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':1},

{'packed':1,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':0},

]

(2) ground\_truth = [1,1,1,1,0,0,0,0]

Listing 8-3: Declaring training and label vectors: Liste 8-3: Eğitim ve etiket vektörlerini beyan etme

Bu örnekte, birlikte eğitim verilerimizi oluşturan iki yapıyı - özellik vektörleri ve bir etiket vektörü - başlatıyoruz. Training\_examples değişkenine (1) atanan özellik vektörleri sözlük biçiminde verilmiştir. Gördüğünüz gibi, iki basit özellik kullanıyoruz. Birincisi, belirli bir dosyanın paketlenip paketlenmediğini temsil eden paketlenmiştir ve ikincisi, dosyanın şifrelenmiş veri içerip içermediğini gösteren contains\_encrypted'dir. Ground\_truth değişkenine (2) atanan etiket vektörü, her eğitim örneğinin kötü niyetli mi yoksa zararsız mı olduğunu temsil eder. Bu kitapta ve genel olarak güvenlik veri bilimcileri arasında, 0 her zaman zararsız ve 1 her zaman kötü niyetli anlamına gelir. Bu durumda etiket vektörü, ilk dört özellik vektörünün kötü niyetli ve ikinci dördünün zararsız olduğunu bildirir.

->Training Your Decision Tree Classifier:Karar Ağacı Sınıflandırıcınızın Eğitimi

Artık eğitim vektörlerimizi ve etiket vektörümüzü açıkladığımıza göre, Liste 8-4'te gösterildiği gibi karar ağacı sınıfının uygunluk yöntemini çağırarak karar ağacı modelimizi eğitelim.

# initialize the vectorizer with the training data

(1)vectorizer.fit(training\_examples)

# transform the training examples to vector form

(2)X = vectorizer.transform(training\_examples)

y = ground\_truth # call ground truth 'y', by convention

Listing 8-4: Initializing the vectorizer class with training data:Liste 8-4: Vektörleştirici sınıfını eğitim verileriyle başlatma

Liste 8-4'teki kod ilk olarak, uygun yöntemini (1) çağırarak Liste 8-2'de başlattığımız vektörleştirici sınıfını başlatır. Burada, fit yöntemi sklearn'a paketlenmiş özellik ile şifrelenmiş özellik ve vektör dizisi indeksleri arasında bir eşleme oluşturmasını söyler. Daha sonra sözlük tabanlı öznitelik vektörlerimizi, vektörleştirici sınıfının dönüştürme yöntemini (2) çağırarak sayısal vektör formuna dönüştürüyoruz. Özellik vektörlerimizi X adlı bir değişkene ve etiket vektörümüzü makine öğrenimi topluluğundaki adlandırma kuralı olan y adlı bir değişkene atadığımızı hatırlayın.

Artık eğitim verilerimizle ayarlandığımıza göre, karar ağacı algılayıcımızı aşağıdaki gibi karar ağacı sınıflandırıcı örneklerinde uygun yöntemi çağırarak eğitebiliriz:

# train the classifier (a.k.a. 'fit' the classifier)

classifier.fit(X,y)

Gördüğünüz gibi, sklearn dedektörünü eğitmek bu kadar basit. Ancak perde arkasında sklearn, önceki bölümde tartıştığımız algoritmanın çizgileri doğrultusunda yeni yazılımın kötü niyetli mi yoksa zararsız mı olduğunu doğru bir şekilde tespit etmek için iyi bir karar ağacı belirleme algoritmik sürecinden geçiyor.

Artık detektörü eğittiğimize göre, bir ikili dosyanın kötü niyetli mi yoksa zararsız mı olduğunu tespit etmek için Liste 8-5'teki kodu kullanalım.

test\_example = (1){'packed':1,'contains\_encrypted':0}

test\_vector = (2)vectorizer.transform(test\_example)

(3) print classifier.predict(test\_vector) # prints [1]

Liste 8-5: Bir ikili dosyanın kötü amaçlı olup olmadığını belirleme

Burada, varsayımsal bir yazılım ikilisi (1) için sözlüğe dayalı bir özellik vektörünü somutlaştırıyoruz, kodumuzda daha önce belirttiğimiz vektörleştiriciyi (2) kullanarak sayısal vektör formuna çeviriyoruz ve sonra oluşturduğumuz karar ağacı algılayıcısını çalıştırıyoruz (3 ) ikili dosyanın kötü amaçlı olup olmadığını belirlemek için. Kodu çalıştırdığımızda sınıflandırıcının yeni ikili dosyanın kötü amaçlı olduğunu "düşündüğünü" (çünkü çıktısı olarak "1" verdiğini) göreceksiniz ve karar ağacımızı görselleştirdiğimizde durumun neden böyle olduğunu göreceksiniz.

->Visualizing the Decision Tree:Karar Ağacını Görselleştirme

Sklearn'ın eğitim verilerimize göre otomatik olarak oluşturduğu karar ağacını Liste 8-6'da gösterildiği gibi görselleştirebiliriz.

# visualize the decision tree

with open((1)"classifier.dot","w") as output\_file:

(2) tree.export\_graphviz(

classifier,

feature\_names=vectorizer.get\_feature\_names(),

out\_file=output\_file

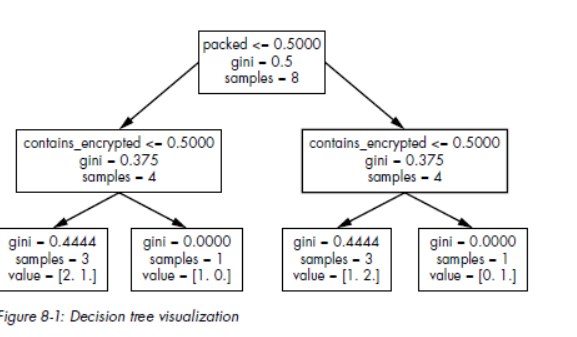
)

import os

os.system("dot classifier.dot -Tpng -o classifier.png")

Listing 8-6: Creating an image file of the decision tree using GraphViz

Burada, sklearn'ın ağaç modülünün sağladığı export\_graphviz () işlevini kullanarak karar ağacımızın bir ağ temsilini yazdığımız classifier.dot (1) adlı bir dosya açıyoruz. Sonra, karar ağacının bir ağ temsilini diske yazan classifier.dot'a bir GraphViz .dot dosyası yazmak için tree.export\_graphviz (2) 'yi çağırırız. Son olarak, Bölüm 6'da karar ağaçları hakkında öğrendiklerinize karşılık gelen bir biçimde karar ağacını görselleştiren bir görüntü dosyası oluşturmak için GraphViz nokta komut satırı programını kullanıyoruz. Bunu çalıştırdığınızda, adlı bir çıktı görüntü dosyası almalısınız. Şekil 8-1'e benzeyen classifier.png



Bu karar ağacı görselleştirmesinin 6. Bölümden aşina olması gerekse de, bazı yeni sözcükler içermektedir. Her kutudaki ilk satır, düğümün hakkında soru sorduğu özelliğin adını içerir (makine öğreniminde, düğümün bu özelliği "ayırdığını splits" söyleriz). Örneğin, ilk düğüm "paketlenmiş packed" özelliğe ayrılır: eğer bir ikili paketli değilse, sol ok boyunca hareket ederiz; aksi takdirde sağ ok boyunca ilerleriz.

Her bir kutudaki ikinci metin satırı, o düğümün, o düğümle eşleşen kötü amaçlı yazılım ve iyi huylu yazılım eğitim örnekleri arasında ne kadar eşitsizlik olduğunu ölçen gini dizinini ifade eder. Gini indeksi ne kadar yüksekse, bu düğümle eşleşen örnekler o kadar çok iyi huylu yazılıma veya kötü amaçlı yazılıma yöneliktir. Bu, her düğümde yüksek bir gini indeksinin iyi olduğu anlamına gelir, çünkü eğitim örnekleri kötü amaçlı veya iyi huylu yazılıma doğru ne kadar çok saparsa, yeni test örneklerinin kötü amaçlı yazılım mı yoksa iyi huylu yazılım mı olduğundan o kadar emin oluruz. Her kutudaki üçüncü satır, yalnızca o düğümle eşleşen eğitim örneklerinin sayısını verir.

Ağacın yaprak düğümlerinde kutudaki metnin farklı olduğunu fark edeceksiniz. Bu düğümler "soru sormaz"; bunun yerine, "bu ikili kod kötü niyetli mi yoksa zararsız mı?" sorusuna yanıt veriyorlar. Örneğin, en soldaki yaprak düğümde "değer = [2. 1.], ”bu, iki iyi huylu eğitim örneğinin bu düğümle eşleştiği (paketlenmemiş ve şifrelenmemiş) ve bir kötü amaçlı yazılım eğitimi örneğinin düğümle eşleştiği anlamına gelir. Yani, bu düğüme ulaşırsak, ikili programın kötü amaçlı yazılım olma olasılığını yüzde 33 olarak atarız (1 kötü amaçlı yazılım örneği / toplam 3 örnek = yüzde 33). Bu kutulardaki gini değeri, doğrudan bu düğümlere giden soruyu böldüğümüzde ikili dosyanın kötü amaçlı yazılım mı yoksa iyi huylu yazılım mı olduğu hakkında ne kadar bilgi kazanıldığını gösterir. Gördüğünüz gibi, karar ağaçlarımızın nasıl algılamalar yaptığını anlamak için sklearn tarafından oluşturulan karar ağaçlarının görselleştirmelerini incelemek faydalı olabilir.

->Complete Sample Code: Tam Örnek Kod:

8-7 Listesi, şimdiye kadar açıkladığım karar ağacı iş akışının tam kodunu gösterir. Kodu parça parça çalıştığımıza göre, bu kod artık sizin için kolayca okunabilir olmalıdır.

#!/usr/bin/python

# import sklearn modules

from sklearn import tree

from sklearn.feature\_extraction import DictVectorizer

# initialize the decision tree classifier and vectorizer

classifier = tree.DecisionTreeClassifier()

vectorizer = DictVectorizer(sparse=False)

# declare toy training data

training\_examples = [

{'packed':1,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':0},

{'packed':1,'contains\_encrypted':1},

{'packed':1,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':1},

{'packed':1,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':0},

{'packed':0,'contains\_encrypted':0},

]

ground\_truth = [1,1,1,1,0,0,0,0]

# initialize the vectorizer with the training data

vectorizer.fit(training\_examples)

# transform the training examples to vector form

X = vectorizer.transform(training\_examples)

y = ground\_truth # call ground truth 'y', by convention

# train the classifier (a.k.a. 'fit' the classifier)

classifier.fit(X,y)

test\_example = {'packed':1,'contains\_encrypted':0}

test\_vector = vectorizer.transform(test\_example)

print `classifier.predict(test\_vector)` # prints [1]

#visualize the decision tree

with open("classifier.dot","w") as output\_file:

tree.export\_graphviz(

classifier,

feature\_names=vectorizer.get\_feature\_names(),

out\_file=output\_file

)

import os

os.system("dot classifier.dot -Tpng -o classifier.png")

Listing 8-7: Complete decision tree workflow sample code: Liste 8-7: Eksiksiz karar ağacı iş akışı örnek kodu

Az önce araştırdığımız örnek makine öğrenimi kötü amaçlı yazılım algılayıcısı, sklearn'ın işlevselliğini nasıl kullanmaya başlayacağınızı gösteriyor, ancak gerçek dünya kötü amaçlı yazılım algılayıcısı için gerekli bazı temel özellikleri eksik. Şimdi gerçek bir kötü amaçlı yazılım algılayıcısının neler gerektirdiğini inceleyelim.

->Building Real-World Machine Learning Detectors with sklearn:Sklearn ile Gerçek Dünya Makine Öğrenimi Dedektörleri Oluşturma

Gerçek dünya dedektörü oluşturmak için, yazılım ikili dosyalarının endüstriyel güçlü özelliklerini industrial-strenght features kullanmanız ve bu özellikleri yazılım ikili dosyalarından software binaries çıkarmak için kod yazmanız gerekir.. Endüstriyel güç özellikleri, ikili dosyaların içeriğini tüm karmaşıklıkları içinde yansıtan özelliklerdir; bu, yüzlerce veya binlerce özelliği kullanmamız gerektiği anlamına gelir. Özellikleri "ayıklamak" derken, bu özelliklerin ikili dosyalar içindeki varlığını tanımlayan kod yazmanız gerektiğini kastediyorum. Ayrıca binlerce eğitim örneği kullanmanız ve uygun ölçekte bir makine öğrenimi modeli eğitmeniz gerekir. Son olarak, sklearn’in daha gelişmiş algılama yaklaşımlarını kullanmanız gerekir, çünkü az önce araştırdığımız basit karar ağacı yaklaşımları yeterli algılama doğruluğu sağlamaz.

\*Real-World Feature Extraction:Gerçek Dünyadan Özellik Çıkarma

Daha önce kullandığım örnek özellikler, örneğin paketlenmiş ve şifrelenmiş veriler içerir, basit oyuncak örnekleridir ve bu iki özellik tek başına hiçbir zaman çalışan bir kötü amaçlı yazılım algılayıcıyla sonuçlanmaz. Daha önce de bahsettiğim gibi, gerçek dünyadaki kötü amaçlı yazılım tespit sistemleri yüzlerce, binlerce ve hatta milyonlarca özellik kullanır. Örneğin, makine öğrenimi tabanlı bir algılayıcı, yazılım ikili dosyalarında bulunan milyonlarca karakter dizesini özellik olarak kullanabilir. Ya da yazılım ikili Taşınabilir Yürütülebilir (PE) başlıklarının değerlerini, belirli bir ikili tarafından içe aktarılan işlevleri veya bunların tümünün bir kombinasyonunu kullanabilir. Bu bölümde yalnızca dizi özellikleriyle çalışacak olsak da, dize özelliklerinden başlayarak makine öğrenimi tabanlı kötü amaçlı yazılım algılamada kullanılan yaygın özellik kategorilerini keşfetmek için biraz zaman ayıralım. !!!!!

-String Features:Dize Özellikleri:

Bir yazılım ikilisinin dize özellikleri, dosyadaki en azından bir miktar minimum uzunlukta olan yazdırılabilir karakterlerin tüm bitişik dizeleridir (bu kitapta bu minimum beş karaktere ayarlanmıştır). Örneğin, bir ikili dosyanın aşağıdaki yazdırılabilir karakter dizilerini içerdiğini varsayalım:

["A", "The", "PE executable", "Malicious payload"]

      Bu durumda, özellik olarak kullanabileceğimiz dizeler "PE çalıştırılabilir" ve "Kötü amaçlı yük" olacaktır çünkü bu iki dizgede beşten fazla karakter vardır.

      Dize özelliklerini sklearn'ın anlayabileceği bir biçime dönüştürmek için onları bir Python sözlüğüne koymamız gerekir. Bunu, gerçek dizeleri sözlük anahtarları olarak kullanarak ve ardından söz konusu ikilinin bu dizeyi içerdiğini belirtmek için değerlerini 1 olarak ayarlayarak yaparız. Örneğin, önceki örnek ikili, {"PE yürütülebilir": 1, "Kötü amaçlı yük": 1} özellik vektörünü alacaktır. Elbette, çoğu yazılım ikili dosyasında yalnızca iki değil, yüzlerce yazdırılabilir dize bulunur ve bu dizeler, bir programın ne yaptığı hakkında zengin bilgiler içerebilir.

Aslında, dizi özellikleri makine öğrenimi tabanlı algılamayla iyi çalışır çünkü yazılım ikili dosyaları hakkında çok fazla bilgi toplarlar. İkili program paketlenmiş bir kötü amaçlı yazılım örneğiyse, muhtemelen birkaç bilgilendirici dizeye sahip olabilir ve bu da kendi başına dosyanın kötü amaçlı olduğu konusunda bir fikir verebilir. Öte yandan, dosyanın kaynaklar bölümünün bazı kısımları paketlenmemiş veya karıştırılmamışsa, bu dizeler dosyanın davranışı hakkında çok şey ortaya koyar. Örneğin, söz konusu ikili program HTTP isteklerinde bulunursa, o dosyanın dize kümesinde "GET% s" gibi dizeleri görmek yaygındır.

     Bununla birlikte, dize özelliklerinin bazı sınırlamaları vardır. Örneğin, gerçek program kodunu içermedikleri için ikili bir programın gerçek mantığı hakkında hiçbir şey yakalayamazlar. Dolayısıyla, dizeler paketlenmiş ikili dosyalarda bile yararlı özellikler olsalar da, paketlenmiş ikili dosyaların gerçekte ne yaptığını açıklamazlar. Sonuç olarak, dizi özelliklerine dayalı dedektörler, paketlenmiş kötü amaçlı yazılımları tespit etmek için ideal değildir.

-Portable Executable(PE)Header Features:

PE başlık özellikleri, her Windows exe ve dll dosyasında bulunan PE başlık meta verilerinden çıkarılır. Bu başlıkların biçimi hakkında daha fazla bilgi için Bölüm 1'e bakın.. Statik program ikili dosyalarından PE özelliklerini çıkarmak için, bu bölümde verilen kodu kullanabilir ve ardından dosya özelliklerini Python sözlük biçiminde kodlayabilirsiniz; burada başlık alanı adı sözlük anahtarı ve alan değeri her bir anahtara karşılık gelen değerdir.PE başlık özellikleri, dizi özelliklerini iyi bir şekilde tamamlar.

Örneğin, dizgi özellikleri, "GET% s" örneği gibi, bir program tarafından yapılan işlev çağrılarını ve ağ aktarımlarını yakalama konusunda genellikle iyi bir iş çıkarırken, PE başlık özellikleri, bir program ikili dosyasının derleme zaman damgası, PE'nin düzeni gibi bilgileri yakalar. bölümleri ve bu bölümlerden hangilerinin yürütülebilir olarak işaretlendiği ve diskte ne kadar büyük oldukları. Ayrıca, bir programın başlangıçta ayırdığı bellek miktarını ve dizgi özelliklerinin yakalamadığı bir program ikili dosyasının diğer birçok çalışma zamanı özelliğini de yakalarlar.

  Paketlenmiş ikili dosyalar ile uğraşırken bile, PE başlık özellikleri paketlenmiş kötü amaçlı yazılımları paketlenmiş iyi huylu yazılımlardan ayırmak için yine de iyi bir iş çıkarabilir. Bunun nedeni, gizleme nedeniyle paketlenmiş ikili dosyaların kodunu göremememize rağmen, kodun diskte ne kadar yer kapladığını ve ikili dosyanın diske nasıl yerleştirildiğini veya bir dizi dosya bölümü üzerinde nasıl sıkıştırıldığını hala görebiliyoruz. Bunlar, bir makine öğrenimi sisteminin kötü amaçlı yazılımları iyi huylu yazılımlardan ayırt etmesine yardımcı olabilecek ayrıntıları anlatıyor. Olumsuz tarafı, PE başlık özellikleri, bir programın çalıştırıldığında yürüttüğü gerçek talimatları veya çağırdığı işlevleri yakalamaz.

-Import Address Table (IAT) Features:

Bölüm 1'de öğrendiğiniz İçe Aktarma Adres Tablosu (IAT) da makine öğrenimi özelliklerinin önemli bir kaynağıdır. IAT, bir yazılım ikili dosyasının harici DLL dosyalarından içe aktardığı işlevlerin ve kitaplıkların bir listesini içerir. Bu nedenle, IAT, önceki bölümde açıklanan PE başlık özelliklerini tamamlamak için kullanabileceğiniz program davranışı hakkında önemli bilgiler içerir.

       IAT'yi makine öğrenimi özelliklerinin kaynağı olarak kullanmak için, her dosyayı bir özellikler sözlüğü olarak temsil etmeniz gerekir; burada içe aktarılan kitaplığın ve işlevin adı anahtar, anahtar ise dosyanın söz konusu belirli içe aktarmayı içerir (örneğin, "KERNEL32.DLL: LoadLibraryA" anahtarı, burada KERNEL32.DLL DLL ve LoadLibraryA işlev çağrısıdır). Bir örnek için IAT özelliklerinin bu şekilde hesaplanmasından elde edilen özellik sözlüğü, bir ikili dosyada gözlemlenen herhangi bir tuşa 1 atayacağımız {KERNEL32.DLL: LoadLibraryA: 1, ...} gibi görünür.

       Kötü amaçlı yazılım algılayıcıları oluşturma deneyimime dayanarak, IAT özelliklerinin nadiren kendi başlarına iyi çalıştığını buldum - bu özellikler program davranışı hakkında yararlı yüksek düzeyli bilgiler yakalamasına rağmen, kötü amaçlı yazılım genellikle IAT'yi kendisini iyi huylu yazılım gibi göstermek için gizler. Kötü amaçlı yazılımlar hiçbir gizleme içermese bile, genellikle iyi huylu yazılımların da içe aktardığı DLL çağrılarını içe aktarır ve bu da, yalnızca IAT bilgilerine dayalı olarak kötü amaçlı yazılım ile iyi huylu yazılım arasında ayrım yapmayı zorlaştırır. Son olarak, kötü amaçlı yazılım paketlendiğinde (sıkıştırılmış veya şifrelenmiş, böylece gerçek kötü amaçlı yazılım kodu yalnızca kötü amaçlı yazılım çalıştırılıp sıkıştırıldıktan sonra görünür hale gelir), IAT, kötü amaçlı yazılımın kullandığı içe aktarmaları değil, yalnızca paketleyicinin kullandığı içe aktarmaları içerir. Bununla birlikte, IAT özelliklerini PE başlık özellikleri ve dizi özellikleri gibi diğer özelliklerle birlikte kullandığınızda, sistem doğruluğunu artırabilirler.

-N-grams:

sipariş ordering düzen order

Şimdiye kadar, herhangi bir sipariş kavramı içermeyen makine öğrenimi özelliklerini öğrendiniz. Örneğin, bir ikilinin belirli bir dizeye sahip olup olmadığını kontrol etmek için dize özelliklerini tartıştık, ancak belirli bir dizenin diskteki ikili dizinin başka bir dizgeden önce mi yoksa sonra mı geldiğini kontrol etmedik.

Ama bazen düzen önemlidir. Örneğin, önemli bir kötü amaçlı yazılım ailesinin yalnızca yaygın olarak kullanılan işlevleri içe aktardığını, ancak bunları çok özel bir sırayla içe aktardığını bulabiliriz, öyle ki işlevleri bu sırayla gözlemlediğimizde, kötü amaçlı yazılım ailesini gördüğümüzü biliyoruz. Bu tür sipariş bilgilerini yakalamak için N-gram adı verilen bir makine öğrenimi konseptini kullanabilirsiniz.

N-gramlar kulağa olduklarından daha egzotik geliyor: sadece özelliklerinizi oluştukları sıraya göre yerleştirmeyi ve ardından sıra üzerinde n uzunluğunda bir pencere kaydırmayı, her adımda pencerenin içindeki özelliklerin sırasını tek bir , toplam özellik.Örneğin, ["how", "now", "brown", "cow"]dizisine sahip olsaydık ve bu diziden uzunluk 2'nin (n = 2) N-gram özelliklerini çıkarmak isteseydik [("how","now"), ("now","brown"),("brown","cow")] özelliklerimiz olarak.

Kötü amaçlı yazılım tespiti bağlamında, bazı veri türleri en doğal olarak N-gram özellikleri olarak temsil edilir. Örneğin, ["inc", "dec", "sub", "mov"] gibi bir ikiliyi kurucu talimatlara ayırdığınızda, bu komut dizilerini yakalamak için N-gram yaklaşımını kullanmak mantıklıdır çünkü bir dizi talimatı temsil etmek, belirli kötü amaçlı yazılım uygulamalarını tespit etmede faydalı olabilir. Alternatif olarak, dinamik davranışlarını incelemek için ikili dosyaları çalıştırırken, ikili dosyaların API çağrıları veya yüksek seviye davranışları dizilerini temsil etmek için N-gram yaklaşımını kullanabilirsiniz.

Bir tür sırayla ortaya çıkan verilerle her çalıştığınızda, makine öğrenimi tabanlı kötü amaçlı yazılım algılama sistemlerinizde N-gram özellikleriyle denemeler yapmanızı öneririm. N'yi hangi sayıya ayarlamanız gerektiğini belirlemek genellikle deneme yanılma gerektirir, bu da N gramlarınızın uzunluğunu belirler. Bu deneme yanılma, test verileriniz üzerinde hangi değerin en iyi doğruluğu sağladığını görmek için n değerini değiştirmeyi içerir. Doğru sayıyı bulduğunuzda, N-gram, program ikili dosyalarının gerçek sıralı davranışlarını yakalamak için güçlü özellikler olabilir ve böylece sistem doğruluğunu artırır.

->Why You Can’t Use All Possible Features:Neden Olası Tüm Özellikleri Kullanamıyorsunuz?

Artık farklı özellik kategorilerinin güçlü ve zayıf yönlerini bildiğinize göre, olası en iyi detektörü oluşturmak için neden tüm bu özellikleri aynı anda kullanamayacağınızı merak ediyor olabilirsiniz. Olası tüm özellikleri kullanmanın iyi bir fikir olmadığı birkaç neden var.

İlk olarak, yeni keşfettiğimiz tüm özellikleri çıkarmak uzun zaman alıyor ve bu da sisteminizin dosyaları ne kadar hızlı tarayabileceğini tehlikeye atıyor. Daha da önemlisi, makine öğrenimi algoritmalarında çok fazla özellik kullanırsanız, bellek sorunlarıyla karşılaşabilirsiniz ve sisteminizin eğitilmesi çok uzun sürebilir. Bu nedenle, sistemlerinizi oluştururken, farklı özellikleri denemenizi ve tespit etmeye çalıştığınız kötü amaçlı yazılım türleri (ve üzerinde yanlış pozitifler üretmekten kaçınmaya çalıştığınız zararlı yazılım türleri) üzerinde iyi çalışanlara odaklanmanızı öneririm.

Ne yazık ki, dizi özellikleri gibi tek bir özellik kategorisine girseniz bile, çoğu makine öğrenimi algoritmasının kaldırabileceğinden daha fazla özelliğe sahip olursunuz. Dizi özelliklerini kullanırken, eğitim verilerinizde oluşan her benzersiz dizi için bir özelliğe sahip olmanız gerekir. Örneğin, eğitim örneği A "merhaba dünya" dizesini içeriyorsa ve eğitim örneği B "merhaba dünya!" Dizesini içeriyorsa, "merhaba dünya" ve "merhaba dünya!" iki ayrı özellik olarak. Bu, binlerce eğitim örneğiyle uğraşırken hızla binlerce benzersiz dizeyle karşılaşacağınız ve sisteminizin bu kadar çok özelliği kullanacağı anlamına gelir.

->Using the Hashing Trick to Compress Features:Özellikleri Sıkıştırmak için Karma İşlemi Hilesini Kullanma

Çok fazla özelliğe sahip olma sorununu aşmak için, özellik karması olarak da bilinen hashing hilesi adı verilen popüler ve basit bir çözümü kullanabilirsiniz. Fikir şu şekildedir: eğitim setinizde bir milyon benzersiz dizgi özelliğine sahip olduğunuzu varsayalım, ancak kullandığınız makine öğrenimi algoritması ve donanımı, tüm eğitim setinde yalnızca 4.000 benzersiz özelliği ele alabilir. Bir milyon özelliği, 4.000 giriş uzunluğundaki bir özellik vektörüne sıkıştırmanın bir yoluna ihtiyacınız var.

Hashing hilesi, her özelliği 4.000 indeksten birine hash ederek bu milyon özelliği 4.000 özellik alanına sığdırıyor. Ardından, orijinal özelliğinizin değerini 4.000 boyutlu özellik vektörünüzdeki o dizindeki sayıya eklersiniz. Tabii ki, özellikler genellikle bu yaklaşımla çarpışır çünkü değerleri aynı boyutta birbirine eklenir. Bu, sistem doğruluğunu etkileyebilir çünkü kullandığınız makine öğrenimi algoritması artık tek tek özelliklerin değerini "göremez". Ancak pratikte, doğruluktaki bu bozulma genellikle çok küçüktür ve özelliklerinizin sıkıştırılmış sunumundan elde ettiğiniz fayda, sıkıştırma işlemi nedeniyle meydana gelen bu hafif bozulmadan çok daha ağır basar.

-Implementing the Hashing Trick:Hashing Hilesini Uygulama

Bu fikirleri daha net hale getirmek için, karma hünerini uygulayan örnek kodda size yol göstereceğim. Burada, algoritmanın nasıl çalıştığını göstermek için bu kodu gösteriyorum; daha sonra, sklearn’ın bu işlevin uygulamasını kullanacağız. Örnek kodumuz bir işlev bildirimiyle başlar:

def apply\_hashing\_trick(feature\_dict, vector\_size=2000):

Apply\_hashing\_trick () işlevi iki parametre alır: orijinal özellik sözlüğü ve hash hilesini uyguladıktan sonra daha küçük özellik vektörünü sakladığımız vektörün boyutu.

Ardından, aşağıdaki kodu kullanarak yeni özellik dizisini oluşturuyoruz

new\_features = [0 for x in range(vector\_size)]

New\_features dizisi, karma hile uygulandıktan sonra özellik bilgilerini depolar. Ardından, listeleme 8-8'deki gibi bir for döngüsü içinde hashing hilesinin anahtar işlemlerini gerçekleştiriyoruz.

for key in (1)feature\_dict:

array\_index = (2)hash(key) % vector\_size

new\_features[array\_index] += (3)feature\_dict[key]

Burada, özellik sözlüğündeki (1) her özelliği yinelemek için bir for döngüsü kullanıyoruz. Bunu yapmak için, önce sözlüğün anahtarlarına hash işlemi uygularız (dizge özellikleri durumunda, bunlar yazılımın ikili dizgelerine karşılık gelir) modulo vector\_size, hash değerleri sıfır ve vector\_size - 1 (2) arasında sınırlandırılır. Bu işlemin sonucunu array\_index değişkeninde saklarız.

Yine de for döngüsü içinde, array\_index dizinindeki new\_feature dizisi girişinin değerini, orijinal özellik dizimizdeki (3) değer kadar artırıyoruz. Dize özellikleri durumunda, özellik değerlerimizin yazılım ikilisinin bu belirli dizeye sahip olduğunu belirtmek için 1 olarak ayarlanması durumunda, bu girişi bir artıracağız. Özelliklerin bir değer aralığına sahip olduğu PE başlık özellikleri durumunda (örneğin, bir PE bölümünün alacağı bellek miktarına karşılık gelir), özelliğin değerini girişe ekleriz.

Son olarak, for döngüsünün dışında, basitçe new\_features sözlüğünü şu şekilde döndürürüz:

return new\_features

Bu noktada sklearn, milyonlarca benzersiz özellik yerine yalnızca binlercesini kullanarak new\_features üzerinde çalışabilir.

-Complete Code for the Hashing Trick:Hashing Hilesi için Tam Kod:

Liste 8-9, artık size aşina olması gereken hash hilesi için tam kodu gösterir.

def apply\_hashing\_trick(feature\_dict,vector\_size=2000):

# create an array of zeros of length 'vector\_size'

new\_features = [0 for x in range(vector\_size)]

# iterate over every feature in the feature dictionary

for key in feature\_dict:

# get the index into the new feature array

array\_index = hash(key) % vector\_size

# add the value of the feature to the new feature array

# at the index we got using the hashing trick

new\_features[array\_index] += feature\_dict[key]

return new\_features

Listing 8-9: Complete code for implementing the hashing trick

Gördüğünüz gibi, özellik hashing hilesini kendi başınıza uygulamak kolaydır ve bunu yapmak, nasıl çalıştığını anlamanızı sağlar. Bununla birlikte, kullanımı kolay ve daha optimize edilmiş sklearn uygulamasını da kullanabilirsiniz.

-Using sklearn’s FeatureHasher

Kendi karma çözümünüzü uygulamak yerine sklearn’in yerleşik uygulamasını kullanmak için önce sklearn’ın FeatureHasher sınıfını şu şekilde içe aktarmanız gerekir:

from sklearn.feature\_extraction import FeatureHasher

Ardından, FeatureHasher sınıfını somutlaştırın:

hasher = FeatureHasher(n\_features=2000)

Bunu yapmak için, karma hilesinin uygulanmasından kaynaklanan yeni dizinin boyutu olarak n\_features'ı bildirirsiniz.

Ardından, karma hilesini bazı özellik vektörlerine uygulamak için, bunları FeatureHasher sınıfının dönüştürme yöntemiyle çalıştırmanız yeterlidir:

features = [{'how': 1, 'now': 2, 'brown': 4},{'cow': 2, '.': 5}]

hashed\_features = hasher.transform(features)

Sonuç, Listeleme 8-9'da gösterilen özellik hashing hilesinin özel uygulamamızla etkili bir şekilde aynıdır. Aradaki fark, iyi korunmuş bir makine öğrenimi kitaplığını kullanmak kendi kodumuzdan daha kolay olduğu için burada sadece sklearn uygulamasını kullanıyor olmamızdır. Örnek kodun tamamı Liste 8-10'da gösterilmektedir.

from sklearn.feature\_extraction import FeatureHasher

hasher = FeatureHasher(n\_features=10)

features = [{'how': 1, 'now': 2, 'brown': 4},{'cow': 2, '.': 5}]

hashed\_features = hasher.transform(features)

Listing 8-10: Implementing FeatureHasher

Devam etmeden önce özellik karması hakkında dikkat edilmesi gereken birkaç nokta var. İlk olarak, tahmin edebileceğiniz gibi, özellik hashing, bir makine öğrenimi algoritmasına aktardığınız özellik bilgilerini gizler, çünkü basitçe aynı bölmeye hash yaptıkları gerçeğine dayanarak özellik değerlerini birbirine ekler. Bu, genel olarak, ne kadar az bölme kullanırsanız (veya bazı sabit sayıdaki bölmelere daha fazla özelliği karma olarak eklerseniz), algoritmanızın o kadar kötü performans göstereceği anlamına gelir. Şaşırtıcı bir şekilde, makine öğrenimi algoritmaları, karma numarası kullanılırken bile iyi çalışabilir ve modern donanımda milyonlarca veya milyarlarca özellik ile başa çıkamadığımız için, güvenlik veri biliminde genellikle özellik karma numarasını kullanmak zorunda kalırız.

Özellik karma numarasının bir başka sınırlaması, modelinizin iç kısımlarını analiz ederken hash ettiğiniz orijinal özellikleri kurtarmayı zorlaştırması veya imkansız hale getirmesidir. Karar ağaçları örneğini ele alalım: Özellik vektörlerimizin her girişine keyfi özellikleri hash ettiğimiz için, belirli bir girişe eklediğimiz özelliklerden hangisinin bu girişte bir karar ağacı algoritmasının bölünmesine neden olduğunu bilmiyoruz. herhangi bir sayıdaki özellik, karar ağacının bu girişi bölmenin iyi bir fikir olduğunu düşünmesine neden olabilirdi. Bu önemli bir sınırlama olmasına rağmen, milyonlarca özelliği yönetilebilir bir sayıya sıkıştırmada özellik hashing hilesinin büyük faydaları nedeniyle güvenlik verisi bilimcileri onunla yaşıyor. Artık gerçek dünyada bir kötü amaçlı yazılım algılayıcısı oluşturmak için gerekli yapı taşlarının üzerinden geçtiğimize göre, uçtan uca bir makine öğrenimi kötü amaçlı yazılım algılayıcısının nasıl oluşturulacağını keşfedelim.

->Building an Industrial-Strength Detector:Endüstriyel Mukavemet Detektörü Oluşturma

Yazılım gereksinimleri perspektifinden bakıldığında, gerçek dünya dedektörümüzün üç şey yapması gerekecektir: eğitim ve tespitte kullanmak için yazılım ikili dosyalarından özellikleri çıkarmak, eğitim verilerini kullanarak kötü amaçlı yazılımları tespit etmek için kendini eğitmek ve yeni yazılım ikili dosyalarında gerçekten algılama gerçekleştirmek. Şimdi bunların her birini yapan ve size hepsinin nasıl birbirine uyduğunu gösterecek olan kodu gözden geçirelim.

Bu bölümde kullandığım koda, bu kitaba eşlik eden koddaki malware\_data\_science / ch8 / code / complete\_detector.py adresinden veya bu kitapla sağlanan sanal makinede aynı konumdan erişebilirsiniz. Tek satırlık bir kabuk komut dosyası olan malware\_data\_science / ch8 / code / run\_complete\_detector.sh, dedektörün kabuktan nasıl çalıştırılacağını gösterir.

\*Extracting Features:Unsurları Çıkarma

Dedektörümüzü oluşturmak için uyguladığımız ilk şey, eğitim ikili dosyalarından özellikleri çıkarmak için koddur (burada standart kodu atlıyorum ve programın temel işlevlerine odaklanıyorum). Özelliklerin ayıklanması, eğitim ikili dosyalarından ilgili verilerin çıkarılmasını, bu özelliklerin bir Python sözlüğünde saklanmasını ve daha sonra, benzersiz özelliklerimizin sayısının engelleyici bir şekilde artacağını düşünürsek, sklearn'ın karma hile uygulamasını kullanarak bunları dönüştürmeyi içerir.

Basitlik uğruna, yalnızca dizgi özelliklerini kullanıyoruz ve hashing numarasını kullanmayı seçiyoruz. 8-11 listesi her ikisinin de nasıl yapılacağını gösterir.

def get\_string\_features((1)path,(2)hasher):

# extract strings from binary file using regular expressions

chars = r" -~"

min\_length = 5

string\_regexp = '[%s]{%d,}' % (chars, min\_length)

file\_object = open(path)

data = file\_object.read()

pattern = re.compile(string\_regexp)

strings = pattern.findall(data)

# store string features in dictionary form

(3) string\_features = {}

for string in strings:

string\_features[string] = 1

# hash the features using the hashing trick

(4) hashed\_features = hasher.transform([string\_features])

# do some data munging to get the feature array

hashed\_features = hashed\_features.todense()

hashed\_features = numpy.asarray(hashed\_features)

hashed\_features = hashed\_features[0]

# return hashed string features

(5) print "Extracted {0} strings from {1}".format(len(string\_features),path)

return hashed\_features

Listing 8-11: Defining the get\_string\_features function

Burada, hedef ikili (1) yolunu ve argümanları olarak sklearn'ın özellik karma sınıfının (2) bir örneğini alan get\_string\_features adında tek bir işlev bildiriyoruz. Ardından, hedef dosyanın dizelerini, minimum uzunluktaki tüm yazdırılabilir dizeleri ayrıştıran bir normal ifade kullanarak çıkarıyoruz. Daha sonra, her dizenin değerini sözlükte 1 olarak ayarlayarak daha fazla işlem için özellikleri bir Python sözlüğünde (3) saklıyoruz, sadece bu özelliğin ikili dosyada mevcut olduğunu belirtir.

Ardından, sklearn’in hashing hile uygulamasını kullanarak hasher'ı çağırarak özellikleri karma hale getiriyoruz. String\_features sözlüğünü bir Python listesine sardığımıza dikkat edin, çünkü sklearn tek bir sözlük yerine dönüştürülecek bir sözlükler listesini iletmemizi gerektirir.

Özellik sözlüğümüze bir sözlük listesi olarak geçtiğimiz için, özellikler diziler listesi olarak döndürülür. Ek olarak, bu kitapta tartışmayacağımız büyük matrisleri işlemek için yararlı olabilecek sıkıştırılmış bir temsil olan seyrek biçimde döndürülürler. Verilerimizi normal bir uyuşuk vektöre geri döndürmemiz gerekiyor.

Verileri normal biçime geri döndürmek için todense () ve asarray () çağırıyoruz ve ardından son özellik vektörümüzü kurtarmak için hasher sonuçları listesindeki ilk diziyi seçiyoruz. Fonksiyondaki son adım basitçe özellik vektörü hashed\_features (5) arayana geri döndürmektir.

\*Training the Detector:Dedektörü Eğitme

Sklearn, makine öğrenimi sistemlerini eğitmenin zor işlerinin çoğunu yaptığından, hedef ikili dosyalarımızdan makine öğrenimi özelliklerini çıkardıktan sonra, bir dedektörü eğitmek yalnızca küçük bir miktar kod gerektirir. Bir dedektörü eğitmek için önce eğitim örneklerimizden özellikleri çıkarmamız ve ardından kullanmak istediğimiz özellik hasherini ve sklearn makine öğrenimi algılayıcısını somutlaştırmamız gerekir (bu durumda rastgele bir orman sınıflandırıcı kullanıyoruz). Daha sonra, örneklerin ikili dosyaları üzerinde eğitmek için detektörde sklearn'ın fit yöntemini çağırmamız gerekir. Son olarak, detektörü ve özellik hasher'ını diske kaydederiz, böylece ileride dosyaları taramak istediğimizde onları kullanabiliriz.

8-12 listesi, dedektörün eğitimi için kodu gösterir.

def (1)get\_training\_data(benign\_path,malicious\_path,hasher):

def (2)get\_training\_paths(directory):

targets = []

for path in os.listdir(directory):

targets.append(os.path.join(directory,path))

return targets

(3) malicious\_paths = get\_training\_paths(malicious\_path)

(4)benign\_paths = get\_training\_paths(benign\_path)

(5) X = [get\_string\_features(path,hasher)

for path in malicious\_paths + benign\_paths]

y = [1 for i in range(len(malicious\_paths))]

+ [0 for i in range(len(benign\_paths))]

return X, y

def (6)train\_detector(X,y,hasher):

classifier = tree.RandomForestClassifier()

(7)classifier.fit(X,y)

(8) pickle.dump((classifier,hasher),open("saved\_detector.pkl","w+"))

Listing 8-12: Programming sklearn to train the detector

Sağladığımız eğitim örneklerinden özellikleri çıkaran get\_training\_data () işlevini (1) bildirerek başlayalım. İşlevin üç bağımsız değişkeni vardır: iyi huylu ikili programların örneklerini içeren bir dizine giden bir yol (benign\_path), kötü niyetli ikili programların örneklerini içeren bir dizine giden yol (kötü niyetli\_yol) ve özellik karması yapmak için kullanılan sklearn FeatureHasher sınıfının bir örneği ( hasher). Daha sonra, bize belirli bir dizinde meydana gelen dosyalar için mutlak dosya yollarının listesini veren yerel bir yardımcı işlev olan get\_training\_paths () (2) 'yi beyan ederiz. Sonraki iki satırda, bize kötü niyetli (3) ve iyi huylu (4) eğitim örnek dizinlerinde oluşan yolların listelerini almak için get\_training\_paths kullanıyoruz.

Son olarak, özelliklerimizi çıkarıyoruz ve etiket vektörümüzü oluşturuyoruz. Bunu, her eğitim örneği dosya yolunda (5) Liste 8-11'de açıklanan get\_string\_features işlevini çağırarak yaparız. Etiket vektörünün her kötü niyetli yol için bir 1'e ve her iyi huylu yol için bir 0'a sahip olduğuna dikkat edin, öyle ki etiket vektöründeki indekslerdeki sayılar, X dizisindeki aynı indekslerdeki özellik vektörlerinin etiketine karşılık gelir. Bu, sklearn'ın özellik ve etiket verilerini beklediği biçimdir ve kitaplığa her özellik vektörünün etiketini söylememizi sağlar.

Artık özellikleri çıkarmayı bitirdiğimize ve özellik vektörümüzü X ve etiket vektörümüz y'yi oluşturduğumuza göre, sklearn'a özellik vektörlerini ve etiket vektörünü kullanarak dedektörümüzü eğitmesini söylemeye hazırız.

Bunu, üç bağımsız değişken alan train\_detector () işlevini (6) kullanarak yapıyoruz: eğitim örneği özellik vektörleri (X), etiket vektörü (y) ve sklearn'ın özellik karma (hasher) örneği. İşlev gövdesinde sklearn detektörü olan tree.RandomForestClassifier'ı başlatıyoruz. Ardından, onu eğitmek için dedektörün yerleştirme yöntemine X ve y'yi geçiririz (7) ve ardından dedektörü ve hasher'ı ileride kullanmak üzere kaydetmek için Python turşu picklemodülünü (8) kullanırız.

\*Running the Detector on New Binaries:Dedektörü Yeni İkililerde Çalıştırma

Şimdi, yeni program ikili dosyalarında kötü amaçlı yazılımları tespit etmek için yeni eğittiğimiz kaydedilmiş algılayıcıyı nasıl kullanacağımızı gözden geçirelim. 8-13 listesi, bunu yapmak için scan\_file () işlevinin nasıl yazılacağını gösterir.

def scan\_file(path):

if not os.path.exists("saved\_detector.pkl"):

print "Train a detector before scanning files."

sys.exit(1)

(1) with open("saved\_detector.pkl") as saved\_detector:

classifier, hasher = pickle.load(saved\_detector)

features = (2)get\_string\_features(path,hasher)

result\_proba = (3)classifier.predict\_proba(features)[1]

# if the user specifies malware\_paths and

# benignware\_paths, train a detector

(4) if result\_proba > 0.5:

print "It appears this file is malicious!",`result\_proba`

else:

print "It appears this file is benign.",`result\_proba`

Listing 8-13: Running the detector on new binaries

Burada, bir dosyanın kötü niyetli mi yoksa zararsız mı olduğunu belirlemek için scan\_file () işlevini açıklıyoruz. Tek argümanı, tarayacağımız ikili dosyaya giden yoldur. Fonksiyonun ilk işi, kaydedilen detektörü ve hasher'ı kaydedildikleri turşu dosyasından yüklemektir (1).

Ardından, Liste 8-11'de tanımladığımız get\_string\_features (2) işlevini kullanarak hedef dosyadan özellikleri çıkarıyoruz.

Son olarak, çıkarılan özellikler göz önüne alındığında, söz konusu dosyanın kötü amaçlı olup olmadığına karar vermek için algılayıcının tahmin yöntemini çağırıyoruz. Bunu, sınıflandırıcı örneğinin predikt\_proba yöntemini (3) kullanarak ve döndürdüğü dizinin ikinci öğesini seçerek yaparız, bu da dosyanın kötü amaçlı olma olasılığına karşılık gelir. Bu olasılık 0,5 veya yüzde 50'nin (4) üzerindeyse dosyanın kötü amaçlı olduğunu söyleriz; aksi takdirde, kullanıcıya zararsız olduğunu söyleriz. Yanlış pozitifleri en aza indirmek için bu karar eşiğini çok daha yüksek bir değere değiştirebiliriz.

\*What We’ve Implemented So Far:Şimdiye Kadar Uyguladıklarımız

8-14 arası liste, bu küçük ölçekli ancak gerçekçi kötü amaçlı yazılım algılayıcısının kodunu bütünüyle gösterir. Her bir parçanın nasıl çalıştığını artık gördüğünüze göre kodun size akıcı bir şekilde okunmasını umuyorum.

#!/usr/bin/python

import os

import sys

import pickle

import argparse

import re

import numpy

from sklearn.ensemble import RandomForestClassifier

from sklearn.feature\_extraction import FeatureHasher

def get\_string\_features(path,hasher):

# extract strings from binary file using regular expressions

chars = r" -~"

min\_length = 5

string\_regexp = '[%s]{%d,}' % (chars, min\_length)

file\_object = open(path)

data = file\_object.read()

pattern = re.compile(string\_regexp)

strings = pattern.findall(data)

# store string features in dictionary form

string\_features = {}

for string in strings:

string\_features[string] = 1

# hash the features using the hashing trick

hashed\_features = hasher.transform([string\_features])

# do some data munging to get the feature array

hashed\_features = hashed\_features.todense()

hashed\_features = numpy.asarray(hashed\_features)

hashed\_features = hashed\_features[0]

# return hashed string features

print "Extracted {0} strings from {1}".format(len(string\_features),path)

return hashed\_features

def scan\_file(path):

# scan a file to determine if it is malicious or benign

if not os.path.exists("saved\_detector.pkl"):

print "Train a detector before scanning files."

sys.exit(1)

with open("saved\_detector.pkl") as saved\_detector:

classifier, hasher = pickle.load(saved\_detector)

features = get\_string\_features(path,hasher)

result\_proba = classifier.predict\_proba([features])[:,1]

# if the user specifies malware\_paths and

# benignware\_paths, train a detector

if result\_proba > 0.5:

print "It appears this file is malicious!",`result\_proba`

else:

print "It appears this file is benign.",`result\_proba`

def train\_detector(benign\_path,malicious\_path,hasher):

# train the detector on the specified training data

def get\_training\_paths(directory):

targets = []

for path in os.listdir(directory):

targets.append(os.path.join(directory,path))

return targets

malicious\_paths = get\_training\_paths(malicious\_path)

benign\_paths = get\_training\_paths(benign\_path)

X = [get\_string\_features(path,hasher) for path in malicious\_paths + benign\_paths]

y = [1 for i in range(len(malicious\_paths))] + [0 for i in range(len(benign\_paths))]

classifier = tree.RandomForestClassifier(64)

classifier.fit(X,y)

pickle.dump((classifier,hasher),open("saved\_detector.pkl","w+"))

def get\_training\_data(benign\_path,malicious\_path,hasher):

def get\_training\_paths(directory):

targets = []

for path in os.listdir(directory):

targets.append(os.path.join(directory,path))

return targets

malicious\_paths = get\_training\_paths(malicious\_path)

benign\_paths = get\_training\_paths(benign\_path)

X = [get\_string\_features(path,hasher) for path in malicious\_paths + benign\_paths]

y = [1 for i in range(len(malicious\_paths))] + [0 for i in range(len(benign\_paths))]

return X, y

parser = argparse.ArgumentParser("get windows object vectors for files")

parser.add\_argument("--malware\_paths",default=None,help="Path to malware training files")

parser.add\_argument("--benignware\_paths",default=None,help="Path to benignware training files")

parser.add\_argument("--scan\_file\_path",default=None,help="File to scan")

args = parser.parse\_args()

hasher = FeatureHasher(20000)

if args.malware\_paths and args.benignware\_paths:

train\_detector(args.benignware\_paths,args.malware\_paths,hasher)

elif args.scan\_file\_path:

scan\_file(args.scan\_file\_path)

else:

print "[\*] You did not specify a path to scan," \

" nor did you specify paths to malicious and benign training files" \

" please specify one of these to use the detector.\n"

parser.print\_help()

Listing 8-14: Basic machine learning malware detector code

Makine öğrenimi tabanlı bir kötü amaçlı yazılım algılayıcı yazmak harikadır, ancak dedektörü etkinliğine güvenerek yerleştirecekseniz performansını değerlendirmek ve iyileştirmek gerekir. Ardından, dedektörünüzün performansını değerlendirmenin farklı yollarını öğrenirsiniz.

->Evaluating Your Detector’s Performance:Dedektörünüzün Performansını Değerlendirme

Sklearn, Bölüm 7'de öğrendiğiniz ROC eğrileri gibi ölçümleri kullanarak algılama sistemlerini değerlendirmeyi kolaylaştıran bir kod içerir. Sklearn kitaplığı ayrıca makine öğrenimi sistemlerini değerlendirmeye özgü ek değerlendirme işlevi sağlar. Örneğin, çapraz doğrulama gerçekleştirmek için sklearn'ın işlevlerini kullanabilirsiniz; bu, dedektörünüzü konuşlandırdığınızda ne kadar iyi çalışacağını tahmin etmek için güçlü bir yöntemdir.

Bu bölümde, dedektörünüzün doğruluğunu gösteren ROC eğrilerini çizmek için sklearn'ı nasıl kullanacağınızı öğreneceksiniz. Ayrıca çapraz doğrulama ve sklearn ile nasıl uygulanacağını da öğrenirsiniz.

\*Using ROC Curves to Evaluate Detector Efficacy:Dedektör Etkinliğini Değerlendirmek için ROC Eğrilerini Kullanma

Alıcı Çalışma Karakteristiği (ROC) eğrilerinin, siz duyarlılığını ayarlarken bir dedektörün gerçek pozitif oranındaki (başarılı bir şekilde algıladığı kötü amaçlı yazılım yüzdesi) ve yanlış pozitif oranındaki (kötü amaçlı yazılım olarak yanlışlıkla işaretlediği kötü niyetli yazılım yüzdesi) değişiklikleri ölçtüğünü hatırlayın. Hassasiyet ne kadar yüksek olursa, o kadar fazla yanlış pozitif alırsınız, ancak tespit oranınız o kadar büyük olur. Duyarlılık ne kadar düşük olursa, yanlış pozitifler o kadar az olur, ancak aynı zamanda daha az algılama alırsınız. Bir ROC eğrisini hesaplamak için, bir tehdit puanı verebilen bir algılayıcıya ihtiyacınız vardır, öyle ki değeri ne kadar yüksekse, ikili dosyanın kötü niyetli olma olasılığı o kadar yüksektir. Sklearn’in karar ağaçları, lojistik regresyon, k-en yakın komşular, rastgele ormanlar ve bu kitapta ele alınan diğer makine öğrenimi yaklaşımları uygulamalarının tümü, bir dosyanın kötü amaçlı yazılım mı yoksa zararlı yazılım mı olduğunu yansıtan bir tehdit puanı oluşturma seçeneği sunar. Bir dedektörün doğruluğunu belirlemek için ROC eğrilerini nasıl kullanabileceğimizi inceleyelim.

-Computing ROC Curves:ROC Eğrilerinin Hesaplanması:

Liste 8-14'te oluşturduğumuz makine öğrenimi algılayıcısı için bir ROC eğrisi hesaplamak için iki şey yapmamız gerekir: birincisi, deneysel bir kurulum tanımlama ve ikincisi, sklearn’in metrik modülünü kullanarak deneyi uygulama. Temel deneysel kurulumumuz için, eğitim örneklerimizi ikiye böleceğiz, böylece ilk yarıyı eğitim için ve ikinci yarıyı ROC eğrisini hesaplamak için kullanacağız. Bu ayrım, sıfırıncı gün kötü amaçlı yazılımları tespit etme sorununu simüle eder. Temel olarak, verileri bölerek, programa şunu söylüyoruz: "Kötü amaçlı yazılımları ve zararlı yazılımları nasıl tanımlayacağımı öğrenmek için kullanacağım bir dizi iyi huylu yazılım ve kötü amaçlı yazılım göster ve ardından bana ne kadar iyi olduğunu test etmek için diğer seti göster Kötü amaçlı yazılım ve zararlı yazılım kavramını öğrendim. " Dedektör, test setinde kötü amaçlı yazılımı (veya zararlı yazılımı) hiç görmediğinden, bu değerlendirme kurulumu, dedektörün gerçekten yeni kötü amaçlı yazılımlara karşı ne kadar iyi çalışacağını tahmin etmenin basit bir yoludur.

Bu bölünmeyi sklearn ile uygulamak basittir. İlk olarak, dedektörün doğruluğunu değerlendirmek istediğimizi belirtmek için dedektör programımızın bağımsız değişken ayrıştırıcı sınıfına aşağıdaki gibi bir seçenek ekliyoruz:

parser.add\_argument("--evaluate",default=False,

action="store\_true",help="Perform cross-validation")

Ardından, programın, Liste 8-15'te gösterilen komut satırı argümanlarını işlediğimiz bölümünde, kullanıcının komut satırı argümanlarına -evaluate eklediği durumu işleyen başka bir elif cümlesi ekliyoruz.

elif args.malware\_paths and args.benignware\_paths and args.evaluate:

(1)hasher = FeatureHasher()

X, y = (2)get\_training\_data(

args.benignware\_paths,args.malware\_paths,hasher)

evaluate(X,y,hasher)

def (3)evaluate(X,y,hasher):

import random

from sklearn import metrics

from matplotlib import pyplot

Listing 8-15: Running the detector on new binaries

Bu kodu ayrıntılı olarak inceleyelim. İlk olarak, sklearn özellik hasherini (1) somutlaştırıyoruz, değerlendirme deneyimiz (2) için ihtiyaç duyduğumuz eğitim verilerini alıyoruz ve ardından eğitim verilerini (X, y) ve özelliği alan eval (3) adlı bir işlevi çağırıyoruz. hasher örneği (hasher) parametreleri olarak ve ardından değerlendirmeyi gerçekleştirmek için ihtiyaç duyduğumuz üç modülü içe aktarır. Detektörü eğitmek için hangi eğitim örneklerinin kullanılacağını ve hangilerinin test edilmesinde kullanılacağını rastgele seçmek için rastgele modülü kullanırız. ROC eğrisini hesaplamak için sklearn'dan metrik modülünü ve ROC eğrisini görselleştirmek için matplotlib'den (veri görselleştirme için fiili standart Python kitaplığı) pyplot modülünü kullanıyoruz.

\*Splitting Data into Training and Test Sets:Verileri Eğitim ve Test Kümelerine Bölme:

Artık eğitim verilerimize karşılık gelen X ve y dizilerini rastgele sıraladığımıza göre, bu dizileri Liste 8-16'da gösterildiği gibi eşit boyutlu eğitim ve test kümelerine bölebiliriz, bu da Listing'de başlayan eval () işlevini tanımlamaya devam eder 8-15.

(1) X, y = numpy.array(X), numpy.array(y)

(2)indices = range(len(y))

(3) random.shuffle(indices)

(4) X, y = X[indices], y[indices]

splitpoint = len(X) \* 0.5

(5) splitpoint = int(splitpoint)

(6) training\_X, test\_X = X[:splitpoint], X[splitpoint:]

training\_y, test\_y = y[:splitpoint], y[splitpoint:]

Liste 8-16: Verileri eğitim ve test setlerine bölme

İlk olarak, X ve y'yi numpy dizilerine (1) dönüştürüyoruz ve sonra X ve y (2) 'deki elemanların sayısına karşılık gelen bir indis listesi oluşturuyoruz. Sonra, bu endeksleri (3) rastgele karıştırıp X ve y'yi bu yeni sıraya (4) göre yeniden sıralarız. Bu, örnekleri rastgele olarak eğitim setimize veya test setimize atamamızı sağlayarak, örnekleri deneysel veri dizinimizde oluştukları sıraya göre bölmememizi sağlar. Rastgele bölünmeyi tamamlamak için, veri kümesini eşit şekilde ikiye bölen dizi dizinini bularak, int () işlevini (5) kullanarak bu noktayı en yakın tam sayıya yuvarlayarak ve ardından X ve y dizilerini gerçekten bölerek dizileri ikiye böleriz. eğitim ve test setlerine (6).

Artık eğitim ve test setlerimize sahip olduğumuza göre, aşağıdakileri kullanarak eğitim verilerini kullanarak karar ağacı algılayıcımızı başlatabilir ve eğitebiliriz:

classifier = RandomForestClassifier()

classifier.fit(training\_X,training\_y)

Daha sonra, bu test örneklerinin kötü amaçlı olma olasılığına karşılık gelen test örneklerimiz için puanlar almak üzere eğitimli sınıflandırıcıyı kullanırız:

scores = classifier.predict\_proba(test\_X)[:,-1]

Burada, sınıflandırıcımızda, test örneklerimizin iyi huylu veya kötü amaçlı yazılım olma olasılığını tahmin eden predikt\_proba () yöntemini çağırıyoruz. Sonra, uyuşmuş indeksleme büyüsünü numpy indexing magic, kullanarak, iyi huylu değil, sadece örneklerin kötü niyetli olma olasılıklarını çıkarıyoruz. Bu olasılıkların gereksiz olduğunu unutmayın (örneğin, bir örneğin kötü niyetli olma olasılığı 0.99 ise, iyi huylu olma olasılığı 0.01'dir, çünkü olasılıklar toplamı 1.00'dir), bu nedenle ihtiyacımız olan tek şey burada kötü amaçlı yazılım olasılığıdır.

\*Computing the ROC Curve:ROC Eğrisinin Hesaplanması:

Artık dedektörümüzü kullanarak kötü amaçlı yazılım olasılıklarını ("puanlar" olarak da adlandırabiliriz) hesapladığımıza göre, ROC eğrimizi hesaplamanın zamanı geldi. Bunu önce sklearn’in metrik modülündeki roc\_curve işlevini şu şekilde çağırarak yapıyoruz:

fpr, tpr, thresholds = metrics.roc\_curve(test\_y, scores)

Roc\_curve işlevi, çeşitli karar eşiklerini veya üzerinde bir yazılım ikili dosyasının kötü niyetli olduğunu düşündüğümüz puan eşiklerini test eder ve bu dedektörü kullanacak olsaydık dedektörün yanlış pozitif oranı ve gerçek pozitif oranının ne olacağını ölçer.

Roc\_curve fonksiyonunun iki argüman aldığını görebilirsiniz: test örneklerimiz için etiket vektörü (test\_y) ve detektörümüzün her eğitim örneğinin ne kadar kötü niyetli olduğuna dair kararını içeren skorlar dizisi. İşlev, üç ilişkili dizi döndürür: fpr, tpr ve eşikler. thresholds

Bu dizilerin tümü eşit uzunluktadır, öyle ki her bir indeksteki yanlış pozitif oranı, gerçek pozitif oran ve karar eşiği birbirine karşılık gelir.

Şimdi hesapladığımız ROC eğrisini görselleştirmek için matplotlib'i kullanabiliriz. Bunu, matplotlib’in pyplot modülündeki plot yöntemini çağırarak yapıyoruz, burada gösterildiği gibi:

pyplot.plot(fpr,tpr,'r-')

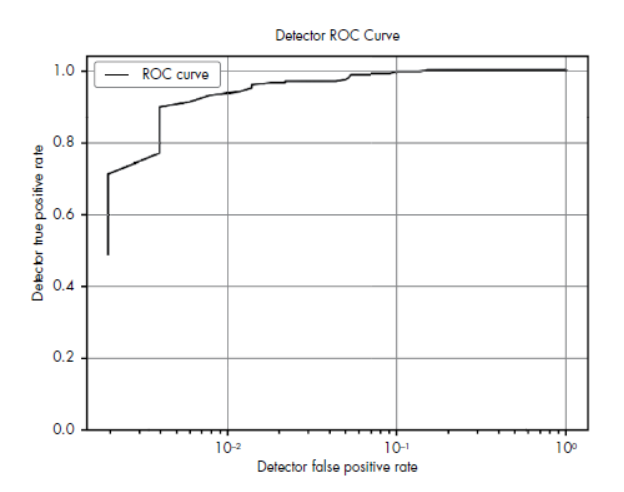
pyplot.xlabel("Detector false positive rate")

pyplot.ylabel("Detector true positive rate")

pyplot.title("Detector ROC Curve")

pyplot.show()

Grafiğin eksenlerini ve başlığını etiketlemek için xlabel, ylabel ve title yöntemlerini ve ardından grafik penceresini açmak için show yöntemini çağırıyoruz. Elde edilen ROC eğrisi Şekil 8-2'de gösterilmektedir.



Şekil 8-2: Dedektörün ROC eğrisini görselleştirme

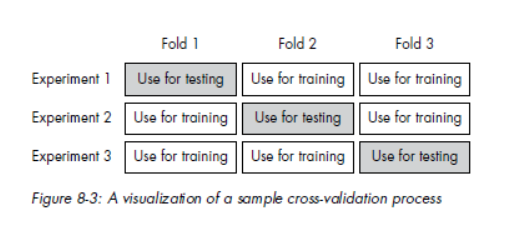
Şekil 8-2'deki grafikten, dedektörümüzün bu kadar basit bir örnek için iyi performans gösterdiğini görebilirsiniz. Yaklaşık yüzde 1 yanlış pozitif oranında (10–2), test setindeki kötü amaçlı yazılım örneklerinin yaklaşık yüzde 94'ünü tespit edebilir. Burada sadece birkaç yüz eğitim örneği üzerinde eğitim veriyoruz; daha iyi bir doğruluk elde etmek için, onu on binlerce, yüz binlerce ve hatta milyonlarca örnek üzerinde eğitmemiz gerekir (ne yazık ki, makine öğrenimini bu dereceye kadar ölçeklendirmek bu kitabın kapsamı dışındadır).

\*Cross-Validation:Çapraz doğrulama

ROC eğrisini görselleştirmek yararlı olsa da, yalnızca bir tane değil, eğitim verilerimiz üzerinde birçok deney gerçekleştirerek dedektörümüzün gerçek dünyadaki doğruluğunu tahmin etmede gerçekten daha iyisini yapabiliriz. Testimizi gerçekleştirmek için eğitim örneklerimizi ikiye böldüğümüzü, dedektörü ilk yarıda eğittiğimizi ve ikinci yarıda test ettiğimizi hatırlayın. Bu gerçekten bizim dedektörümüzün yetersiz bir testidir. Gerçek dünyada, bu belirli test örneklerindeki doğruluğumuzla değil, daha önce görülmemiş yeni kötü amaçlı yazılımlardaki doğruluğumuzla ölçülmeyeceğiz. Dağıttıktan sonra nasıl performans göstereceğimize dair daha iyi bir fikir edinmek için, tek bir test verisi kümesi üzerinde birden fazla deneme yapmamız gerekir; Birçok test setinde birçok deney gerçekleştirmemiz ve doğruluktaki genel eğilimi anlamamız gerekir.

Bunu yapmak için çapraz doğrulama kullanabiliriz. Çapraz doğrulamanın arkasındaki temel fikir, eğitim örneklerimizi birkaç katlamaya bölmektir (burada üç kat kullanıyorum, ancak daha fazlasını kullanabilirsiniz). Örneğin, 300 örneğiniz varsa ve bunları üçe bölmeye karar verdiyseniz, ilk 100 örnek ilk kat, ikinci 100 örnek ikinci kat ve üçüncü 100 örnek üçüncü kat olacaktır.

Sonra üç test yaparız. İlk testte, sistemi 2. ve 3. katlar üzerinde eğitiyoruz ve sistemi 1. kat üzerinde test ediyoruz. İkinci testte, bu işlemi tekrarlıyoruz, ancak sistemi 1. ve 3. katlarda eğitiyor ve sistemi 2. kat üzerinde test ediyoruz. üçüncü test, şimdiye kadar muhtemelen tahmin edebileceğiniz gibi, sistemi 1. ve 2. katlar üzerinde eğitiyoruz ve sistemi 3. kat üzerinde test ediyoruz. Şekil 8-3, bu çapraz doğrulama sürecini göstermektedir.



Şekil 8-3: Örnek çapraz doğrulama sürecinin görselleştirilmesi

Sklearn kitaplığı, çapraz doğrulama uygulamayı kolaylaştırır. Bunu yapmak için, 8-15 Listesindeki değerlendirme fonksiyonumuzu cv\_evaluate olarak yeniden yazalım.

def cv\_evaluate(X,y,hasher):

import random

from sklearn import metrics

from matplotlib import pyplot

from sklearn.cross\_validation import KFold

Cv\_evaluate () işlevini, ilk değerlendirme işlevimizi başlattığımız şekilde başlatırız, ancak burada ayrıca KFold sınıfını sklearn’in cross\_validation modülünden içe aktarırız. K katı çapraz doğrulama veya kısaca KFold, az önce bahsettiğim çapraz doğrulama türüyle eş anlamlıdır ve çapraz doğrulama yapmanın en yaygın yoludur.

Ardından, eğitim verilerimizi numpy'nin gelişmiş dizi indekslemesini kullanabilmemiz için numpy dizilerine dönüştürüyoruz:

X, y = numpy.array(X), numpy.array(y)

Aşağıdaki kod aslında çapraz doğrulama sürecini başlatır:

fold\_counter = 0

for train, test in KFold(len(X),3,ushuffle=True):

(2)training\_X, training\_y = X[train], y[train]

test\_X, test\_y = X[test], y[test]

İlk olarak, ilk parametre olarak sahip olduğumuz eğitim örneklerinin sayısını ve ikinci argüman olarak kullanmak istediğimiz kat sayısını ileterek KFold sınıfını somutlaştırıyoruz. Üçüncü argüman, shuffle = True (1), sklearn'a eğitim verilerimizi üç bölüme ayırmadan önce rastgele sıralamayı söyler. KFold örneği aslında her yinelemede farklı bir eğitim veya test örneği ayırma sağlayan bir yineleyicidir. For döngüsü içinde, eğitim örneklerini ve test örneklerini karşılık gelen öğeleri içeren training\_X ve training\_y dizilerine (2) atarız.

Eğitim ve test verilerini hazırladıktan sonra, bu bölümde daha önce yapmayı öğrendiğiniz gibi, RandomForestClassifier'ı başlatmaya ve eğitmeye hazırız:

classifier = RandomForestClassifier()

classifier.fit(training\_X,training\_y)

Son olarak, bu belirli kat için bir ROC eğrisi hesaplıyoruz ve ardından bu ROC eğrisini temsil eden bir çizgi çiziyoruz:

scores = classifier.predict\_proba(test\_X)[:,-1]

fpr, tpr, thresholds = metrics.roc\_curve(test\_y, scores)

pyplot.semilogx(fpr,tpr,label="Fold number {0}".format(fold\_counter))

fold\_counter += 1

Henüz grafiği görüntülemek için matplotlib show yöntemini çağırmadığımızı unutmayın. Bunu tüm kıvrımlar değerlendirildikten sonra yapıyoruz ve aynı anda üç çizgiyi de göstermeye hazırız. Önceki bölümde yaptığımız gibi, eksenlerimizi etiketleriz ve grafiğe şöyle bir başlık veririz:

pyplot.xlabel("Detector false positive rate")

pyplot.ylabel("Detector true positive rate")

pyplot.title("Detector Cross-Validation ROC Curves")

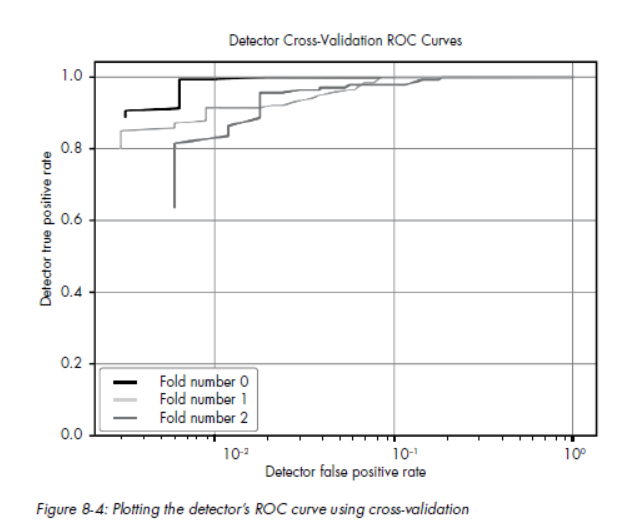
pyplot.legend()

pyplot.grid()

pyplot.show()

Elde edilen ROC eğrisi Şekil 8-4'te gösterilmektedir.

Gördüğünüz gibi, sonuçlarımız her katmanda benzerdi, ancak kesinlikle bazı farklılıklar var. Üç çalışma boyunca tespit oranımız (gerçek pozitif oran), yüzde 1'lik bir yanlış pozitif oranında ortalama yüzde 90'dır. Üç çapraz doğrulama deneyini de hesaba katan bu tahmin, dedektörümüzün performansının, verilerimiz üzerinde yalnızca bir deney yaptığımızda elde edeceğimizden daha doğru bir tahminidir; bu durumda, eğitim ve test için kullandığımız örnekler biraz rastgele bir sonuca yol açacaktır. Daha fazla deney yürüterek, çözümümüzün etkinliği hakkında daha sağlam bir fikir edinebiliriz.



Çok az miktarda veri üzerinde eğitim verdiğimiz için bu sonuçların harika olmadığını unutmayın: birkaç yüz kötü amaçlı yazılım ve iyi huylu yazılım örneği. Büyük ölçekli makine öğrenimi kötü amaçlı yazılım algılama sistemlerini eğittiğimiz günlük işimde, genellikle yüz milyonlarca örnek üzerinde eğitim alıyoruz. Kendi kötü amaçlı yazılım algılayıcınızı eğitmek için yüz milyonlarca örneğe ihtiyacınız yok, ancak gerçekten iyi bir performans elde etmeye başlamak için en az on binlerce örnekten oluşan veri kümelerini bir araya getirmek isteyeceksiniz (örneğin, bir Yüzde 0.1 yanlış pozitif oranı).

\*Next Step:

Şimdiye kadar, yazılım ikili dosyalarının eğitim veri kümesinden özellikleri çıkarmak için Python ve sklearn'ı nasıl kullanacağımı ve ardından karar ağacı tabanlı bir makine öğrenimi yaklaşımını eğitip değerlendirmeyi ele aldım. Sistemi iyileştirmek için, yazdırılabilir dizi özelliklerinin dışında veya bunlara ek özellikler (örneğin, PE başlığı, talimat N gram veya daha önce tartışılan Adres Tablosunu İçe Aktar özellikleri) kullanabilir veya farklı bir makine öğrenimi algoritması kullanabilirsiniz.!!!!YAPMAMA GEREKEN

Dedektörü daha doğru hale getirmek için, sklearn'ın RandomForestClassifier'ın (sklearn.ensemble.RandomForestClas ifier) ​​ötesine geçerek diğer sınıflandırıcıları denemenizi tavsiye ederim. Önceki bölümden, rastgele orman dedektörlerinin de karar ağaçlarına dayandığını, ancak tek bir karar ağacı yerine, inşa edilme şekillerini rastgele hale getirerek birçok karar ağacı inşa ettiklerini hatırlayın. Yeni bir dosyanın kötü amaçlı yazılım mı yoksa zararlı yazılım mı olduğunu belirlemek için, bu karar ağaçlarının her biri, ortalama sonucu elde etmek için bunları toplayarak ve toplam ağaç sayısına bölerek birleştirdiğimiz bağımsız kararlar verir.

Ayrıca, lojistik regresyon gibi sklearn tarafından sağlanan diğer algoritmaları da kullanabilirsiniz. Bu algoritmalardan herhangi birini kullanmak, bu bölümde tartışılan örnek kodda arama yapmak ve değiştirmek kadar basit olabilir. Örneğin, bu bölümde karar ağacımızı aşağıdaki gibi başlatıyor ve eğitiyoruz:

classifier = RandomForestClassifier()

classifier.fit(training\_X,training\_y)

Ancak bu kodu şununla değiştirebilirsiniz:

from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression()

classifier.fit(training\_X,training\_y)

Bu değiştirme, karar ağacı tabanlı bir algılayıcı yerine bir lojistik regresyon algılayıcısı sağlar. Bu Lojistik Regresyon detektörünün yeni bir çapraz doğrulama tabanlı değerlendirmesini hesaplayarak ve bunu Şekil 8-4'teki sonuçlarla karşılaştırarak, hangisinin daha iyi çalıştığını belirleyebilirsiniz.

->ÖZET:

Bu bölümde, makine öğrenimi tabanlı kötü amaçlı yazılım algılayıcıları oluşturmanın püf noktalarını öğrendiniz. Özellikle, makine öğrenimi için yazılım ikili dosyalarından özelliklerin nasıl çıkarılacağını, karma hile kullanarak bu özelliklerin nasıl sıkıştırılacağını ve bu çıkarılan özellikleri kullanarak makine öğrenimi tabanlı kötü amaçlı yazılım algılayıcılarının nasıl eğitileceğini öğrendiniz. Ayrıca, bir dedektörün algılama eşiği ile doğru ve yanlış pozitif oranları arasındaki ilişkiyi incelemek için ROC eğrilerini nasıl çizeceğinizi öğrendiniz. Son olarak, bu bölümde kullanılan dedektörü geliştirmek için çapraz doğrulama, daha gelişmiş bir değerlendirme kavramı ve diğer olası uzantıları öğrendiniz. Bu, bu kitabın sklearn kullanarak makine öğrenimi tabanlı kötü amaçlı yazılım tespitiyle ilgili tartışmasının sonucudur. Derin öğrenme yöntemleri veya yapay sinir ağları olarak bilinen başka bir dizi makine öğrenimi yöntemini 10. ve 11. Bölümlerde ele alacağız. Artık, kötü amaçlı yazılım tanımlama bağlamında makine öğrenimini etkili bir şekilde kullanmak için gerekli temel bilgilere sahipsiniz. Makine öğrenimi hakkında daha fazla bilgi okumanızı tavsiye ederim. Bilgisayar güvenliği birçok yönden bir veri analizi sorunu olduğu için, makine öğrenimi güvenlik endüstrisinde kalmak için burada ve yalnızca kötü amaçlı ikili dosyaları tespit etmede değil, aynı zamanda ağ trafiğinde, sistem günlüklerinde ve diğer bağlamlarda kötü niyetli davranışları tespit etmede de faydalı olmaya devam edecek. .

Sonraki bölümde, çok sayıda kötü amaçlı yazılım örneği arasındaki benzerlikleri ve farklılıkları hızlı bir şekilde anlamamıza yardımcı olabilecek kötü amaçlı yazılım ilişkilerini görselleştirmeye derinlemesine bakacağız.

12.BÖLÜME BİR BAK NASIL KULLANCAĞIMA DAİR