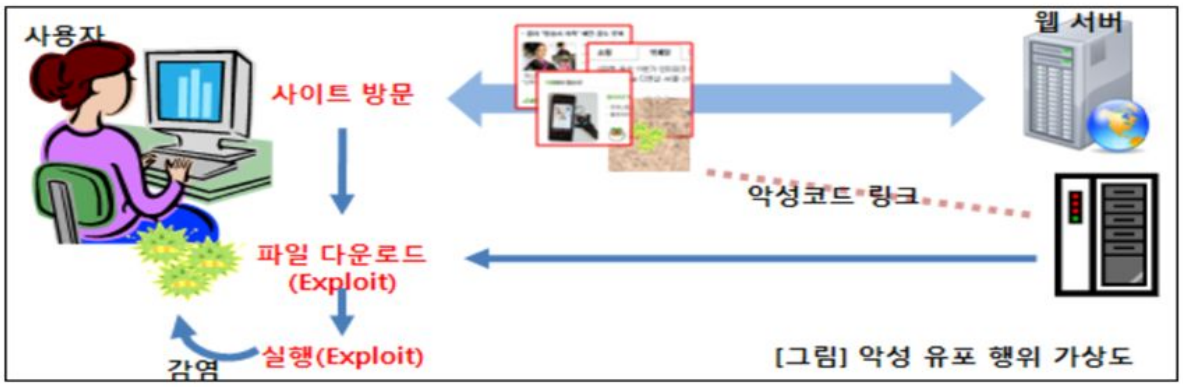
Malicious URL Detector

1. 프로젝트 개요 및 목적

본 프로젝트에서 일반적인 악성 코드의 감염 경로는 대부분이 url을 통하여 안전하지 않은 웹사이트로 접속 혹은 스팸 메일에 있는 악성 프로그램 다운 후 실행 같은 기초적인 것에 의해 발생한다고 한다. 가장 기초적이다보니, 사용자가 웹 서핑을 하면서 악성 URL 등을 무의식적으로 누르기 쉽다. 그렇기 때문에 사용자에게 경각심을 알려주기 위한 방안으로 설계를 시작하였다.

사용자가 웹사이트를 이동할 때마다 해당 페이지에 존재하는 url들 중에 malicious url이 존재하는 지를 판단해야 하고, 이를 위해서는 기존에 존재하는 url 데이터셋을 통해 학습이 되어있는 분류 모델이 준비되어 있어야 한다.

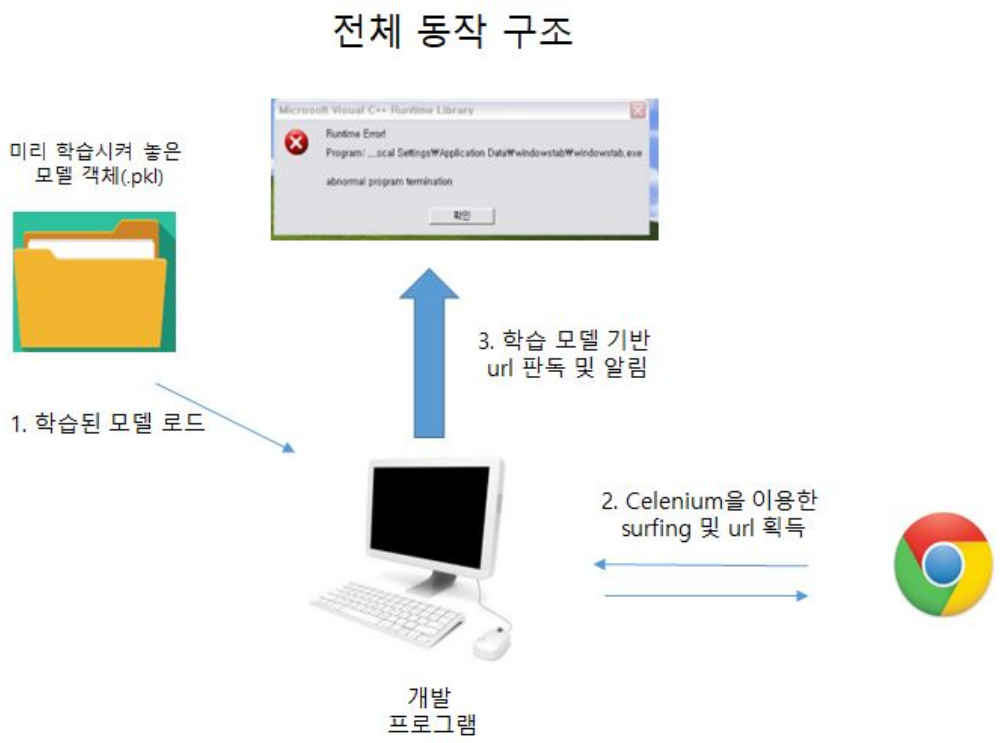


<그림 1 > 기존 악성코드 감염 경로

2. 동작 구조

처음에는 기계학습을 시키면 프로세스가 어떻게 동작을 하는지 , 그리고 어떤 방식으로 결과를 출력하는지 잘 알지 못하였다. 그렇기 때문에 하나는 계속 머신 러닝을 돌리고, 다른 하나는 머신 러닝에 데이터를 넘겨주는 방식으로 시스템을 구성하였다.
하지만 공부를 하며 기계학습을 진행시키다 보니 해당 방식이 비효율적이라는 것을 발견하여서 처음에는 프로세스를 2개 -> 1개로 변경을 하였다. 그리고 학습을 시키고 프로세스를 끌 경우 모델을 다시 학습을 시켜야하는 번거로움이 있었다. 하지만 계속 공부를 하다보니 이러한 객체를 저장하는 모듈인 Pickle 모듈에 대해서도 알게 되었고 최종적인 시스템 구조를 다음 그림과 같이 변경하였다.

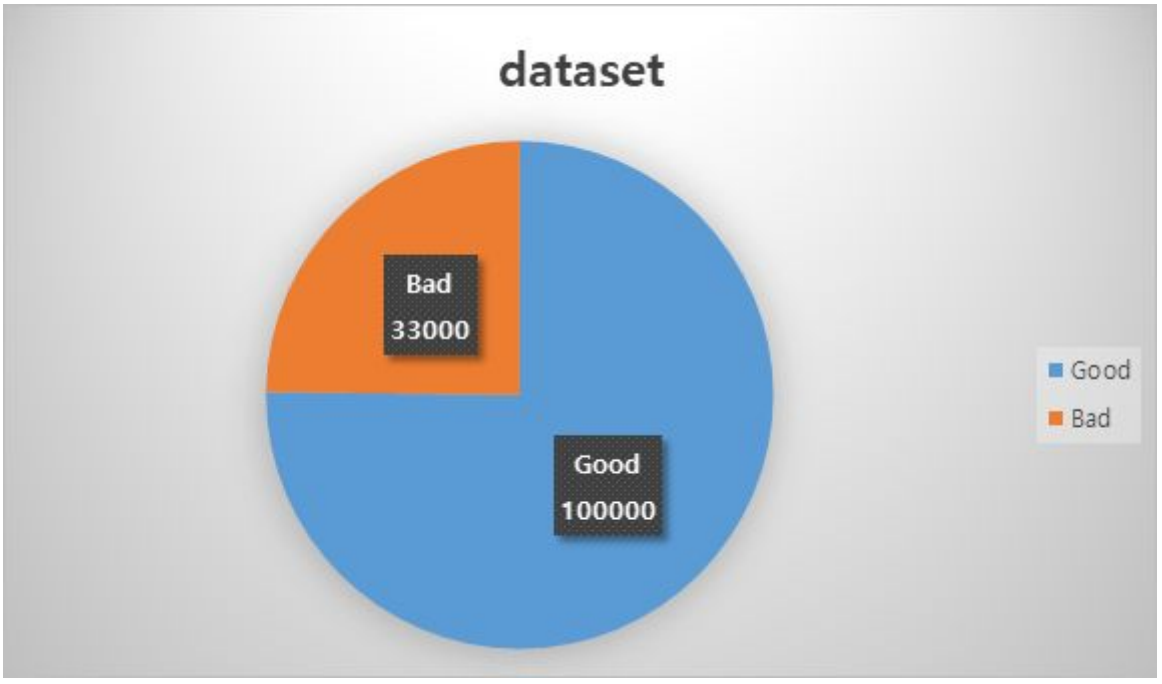
사진상에는 잘 보이지 않지만, 단순히 과정이 간단해 졌다는 것만 이해하면 될 것 같다. Pickle 모듈을 사용하니 학습 시킨 모델을 저장하고 로드 함으로써 모델 학습에 걸리는 시간을 아낄 수 있게 되었다. 처음 시작에서는 모델 선정에 있어서 학습 속도를 우선적으로 고려하였지만, 이러한 속도에 대한 것을 제쳐둘 수 있었다. 이후부터는 모델선정에 있어서 훈련속도는 배제하였다.



< 그림 2 > 개발 계획 및 동작 구조

3. 데이터 셋 및 전처리 과정

프로젝트에 사용할 데이터 셋은 케글을 통해 구했다 . 약 42만개의 데이터로 구성되어 있으며 일반적인 사이트는 good, 악성 URL은 bad으로 레이블링이 되어있는 데이터였다.
이 데이터 셋이 크기도 하고, 의미 없는 데이터도 많다고 판단하여 중복되는 url등을 삭제하여 그 중 악성 URL 은 약 33000개, 그리고 일반적인 URL은 약 10만개로 비율은 다음 표와 같이 나타낼 수 있었다.



< 그림 3 > Data set 레이블 비율

이러한 데이터를 기계 학습에 이용하기 위해서 몇가지 전처리 방식이 존재하였다. 사이킷런에서는 이러한 데이터의 전처리를 위해서 몇가지 방식을 제공한다. 우리는 그 중 CouterVectorizer , TfidfVectorizer, HashingVectorizer 방식을 이용하기로 결정하였다. 우리조는 기계 학습을 모두 처음 접했기 때문에 어떠한 벡터를 사용해야 하는지 등에 대해서 잘 알지 못하였다. 그렇기 때문에 이러한 방식을 같은 모델에 모두 적용해 보기로 하였다. 다음은 각 벡터를 적용하기 위해서 사용했던 전처리 과정이다. 먼저 CounterVectorizer 와 tfidVectorizer 방식의 경우 단어의 카운팅에 기반하고 있기 때문에 우리가 가지고 있는 데이터(URL)를 단어 단위로 쪼개어서 저장하였다.

```
def makeTokens(f):
    tkns_BySlash = str(f.encode('utf-8')).split('/')
    total_Tokens = []
    tkns_BySlash = f.split("\n")
    for i in range(len(tkns_BySlash)):
        total_Tokens = total_Tokens+re.findall(r"[\w']+", tkns_BySlash[i])
    total_Tokens = list(set(total_Tokens))
    return total_Tokens
```

mundovirtualhabbo.blogspot.com/2009_01_01_archive.html	bad
creditgratosse.blogspot.com/2008/01/connnectoi_03.html	bad
aijcs.blogspot.com/2005/03/colourful-life-of-aij.html	bad
tudu-free.blogspot.com/2008/02/jogos-java-aplicativos.html#foote	bad
floridarentfinders.com/uploads/ws/css/www.paypal.com/cgi-bin/w	bad
paypollar.com.p12.hostingprod.com/wbsc.php	bad
01453.com/	good
015fb31.netsolhost.com/bosstweed.html	good
02bee66.netsolhost.com/lincolnhomepage/	good
02ec0a3.netsolhost.com/getperson.php?personID=I4920&tree=ncs	good
032255.com/	good
05minute.com/	good

<그림 4 > 데이터 전처리 코드(좌) , 데이터 셋 중 일부(우)

다음 코드에 대해서 간단히 설명하자면. 우리가 가지고 있는 Url 을 특수문자를 기준으로 단어 단위로 나누는 과정이다. 이 후, 이렇게 생성한 토큰을 통해서 벡터를 생성하였다.

```
import pandas as pd
import numpy as np
from urllib.parse import urlparse
data = pd.read_csv("data.csv")
url_data = []
label_data = []
url_data2 = []
# domain이 동일한 url제거
for idx, url in enumerate(data['url']):
    tmp = url[0:10]
    domain = tmp
    if domain in url_data:
        continue  # 동일 domain 있으면 continue
    else:  # 동일 domain 아니면 저장
        url_data.append(domain)
        url_data2.append(url)
        label_data.append(data['label'][idx])

url_df = pd.DataFrame(url_data2, columns=['url'])
label_df = pd.DataFrame(label_data, columns=['label'])

result_data = pd.concat([url_df, label_df], axis=1)
print(result_data)
result_data.to_csv("domain_data.csv", index=False, columns=['url', 'label'], mode='w')
```

< 그림 5 > Url Data 전처리

또한 중복되는 도메인이 많아 과대적합될 가능성이 크므로 위와 같은 과정을 통해 URL의 앞 10 글자가 동일 할 시 겹치는 URL의 경우 다 삭제했다.

a. TF-idf Vector

```
vectorizer = TfidfVectorizer(tokenizer=makeTokens)
X = vectorizer.fit_transform(url_list)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = TEST_SIZE, random_state = RS)
```

< 그림 6 > TF-idf Vector

b. Count Vector

```
cVec = CountVectorizer(tokenizer=makeTokens)
count_X = cVec.fit_transform(url_list)
cX_train, cX_test = train_test_split(count_X, test_size = TEST_SIZE, random_state = RS)
```

< 그림 7 > Counter Vector

c. Hashing Vector

```
hVec = HashingVectorizer(n_features=300000)
hash_X = hVec.fit_transform(url_list)
hX_train, hX_test = train_test_split(hash_X, test_size = TEST_SIZE, random_state=RS)
```

< 그림 8 > Hashing Vector

일련의 전처리 과정 후, 이러한 벡터들을 기준으로 여러 모델들에 적용을 해보았다. 해당 벡터에 따라서 같은 모델이라도 다른 결과값이 나오는 것을 볼 수 있었다. 즉, 데이터 셋에 맞는 모델 선정과 벡터라이징의 중요성을 알 수 있었다.

다음 표는 Logistic Regression에 각 벡터를 이용한 결과 값이다.

	학습 시간(Sec)	정확도(%)
Count Vector	5.6 s	96.8%
Hashing Vector	45.5 s	94.15%
Tf-idf	5.6 s	96.45%

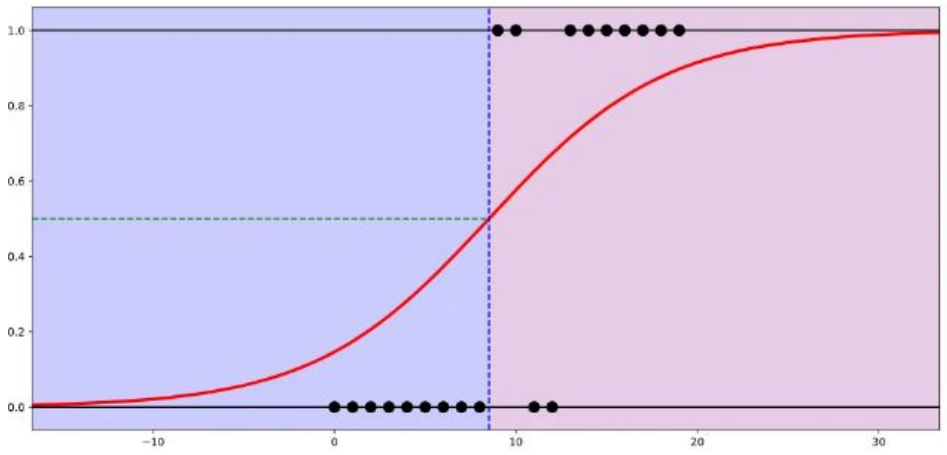
이 중, 최종적으로 Tf-idf 방식을 사용하기로 했다. 악성 Url을 판단하는 기준에 Url의 단어가 큰 기준이 될 것이라고 생각하여서 Tf-idf, Counter Vector를 먼저 우선시 하였고, 두 벡터는 방식이 비슷하나 단순히 단어를 카운트하기만 하는 Count Vector 보다는 추가적으로 단어에 가중치를 두는 Tf-idf가 조금 더 괜찮은 전처리라고 생각을 하였기 때문이다.

4. 후보 모델 선정

후보 모델을 선정하는 데에 있어서 기준은 단순하게 우리가 가진 데이터셋에 얼마나 좋은 성능을 보이는 지와 데이터 셋을 예측하는 속도를 보고 결정해보았다.

4-1 Logistic Regression

로지스틱 회귀(Logistic Regression)는 회귀를 사용하여 데이터가 어떤 범주에 속할 확률을 0 에서 1 사이의 값으로 예측하고 추정 확률이 50 이상일 경우 해당 클래스에 속하는 것으로 분류해주는 이진 분류 모델이다. 아래의 그림과 같이 특정 x값을 기준으로 x값의 범주에 따라 y의 값을 결정한다. 이렇게 데이터가 2개의 범주 중 하나에 속하도록 결정하는 것을 이진 분류라고 한다. 이번 프로젝트는 특정 url이 malicious url인지 아닌지를 판단해주는 것이기 때문에 로지스틱 회귀가 알맞은 모델일 거라 생각했다.



< 그림 9 > 로지스틱 회귀 예시

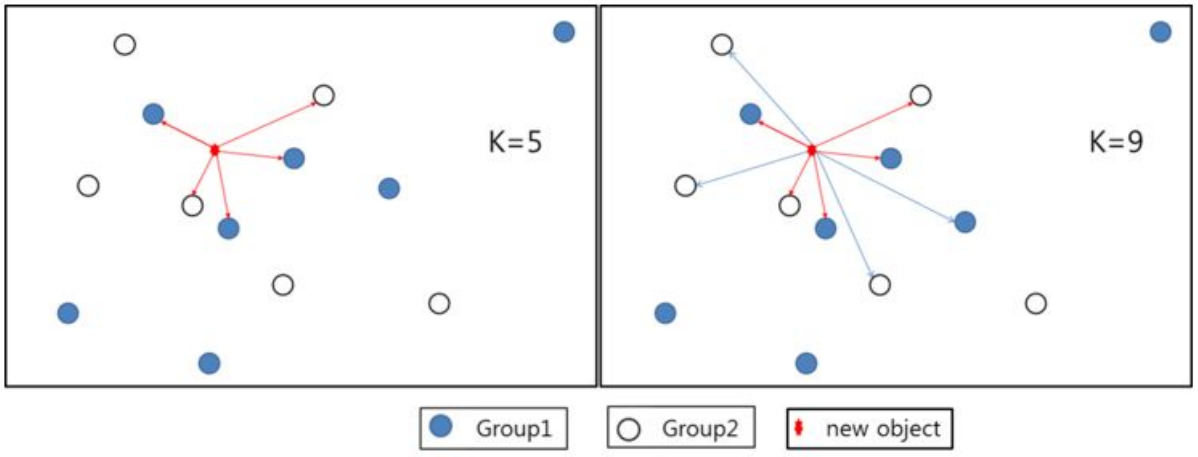
전처리 과정에서 TF-idf를 사용하여 만들어낸 훈련 데이터를 통해 로지스틱 회귀를 사용하여 다음과 같은 결과를 얻어 낼 수 있었다.

	로지스틱 - Tf-idf 벡터
훈련 데이터 정확도	93.9%
테스트 셋 정확도	94%

예상대로 로지스틱은 우리가 가진 url 데이터셋에 대해서 우수한 성능을 보였다. 하지만 훈련 데이터와 테스트 데이터에 대한 성능이 다소 높아 데이터에 대한 과대적합이 우려되었다. 이는 규제를 가하여 일반화 성능을 높이면 해결할 수 있는 문제이므로 잘만 훈련시킨다면 이번 프로젝트에 사용될 모델이 될 수 있을 것 같아서 로지스틱 회귀를 선정했다.

4-2 KNN (Nearest Neighbor Analysis)

KNN에서는 근접한 객체들을 이웃(Neighbor)이라고 하고, 새로운 객체가 나타났을 때 기존의 객체들과의 거리를 계산하여 가장 근접한 객체 k개의 group을 비교해 가장 많이 나타나는 group을 새로운 객체의 group으로 설정한다. 아래의 경우 k=5 일 때, 새로운 객체의 근처에 Group1의 개수가 3개이므로 새로운 객체는 그룹1에 속하게 된다.



< 그림 10 > KNN 예시

KNN의 경우 훈련 단계가 빠르고 특별한 모델을 생성 하지않아 단순하며 효율적이지만 분류 단계가 느리고 많은 메모리가 필요하다는 단점이 있다. 그 예로 훈련 데이터를 테스트 하던 도중 메모리가 부족하여 아래와 같이 에러가 나 정확도를 구하지 못하였다. 그래서 결과를 보기 위해서 데이터 셋을 많이 낮추어서 돌려봤다.

```
MemoryError: Unable to allocate 1023. MiB for an array with shape
(1002, 133845) and data type float64
```

```
2번 모델 (KNN 알고리즘 Tfid 벡터)
test accuracy:  0.8731343283582089
```

< 그림 11 > KNN모델 적용 결과

이처럼 훈련에 있어서는 많은 장점을 가지고 있지만 데이터를 분류하는 데에는 속도가 많이 느리고 많은 메모리가 필요하다는 단점으로 인해서 KNN은 선정대상에서 제외되었다.

4-3. Naive bayes (MultinomialNB)

나이브 베이즈 분류기는 선형 모델과 매우 유사하다. 하지만 선형 분류기보다 훈련 속도가 빠르지만 일반화 성능이 조금 떨어진다. 나이브 베이즈 분류기가 효과적인 이유는 각 특성을 개별로 취급해 파라미터를 학습하고 각 특성에서 클래스별 통계를 단순하게 취합하기 때문이다. 나이브 베이즈에는 여러가지 분류기가 있는데 MultinomialNB를 택했다. MultinomialNB는 alpha 매개변수가 있어서 모든 특성에 +의 값을 가진 가상의 데이터 포인트를 alpha개 추가한다. 이는 통계 데이터를 완만하게 만들고 모델의 복잡도는 낮아진다.

나이브 베이즈는 모형이 단순하고 계산이 효율적이라는 장점이 있지만 예측변수의 범주가 학습용 데이터에서 존재하지 않는 경우 이러한 예측 변수의 범주를 갖는 새로운 레코드는 0의 확률값을 갖는다고 가정하기 때문에 이 프로젝트의 모델에는 적합하지 않다고 판단하여 사용하지 않기로 했다.

	나이브 베이즈 - TF-idf
훈련 데이터 정확도	99.9%
테스트 셋 정확도	95.6%

4-4. Linear Regression

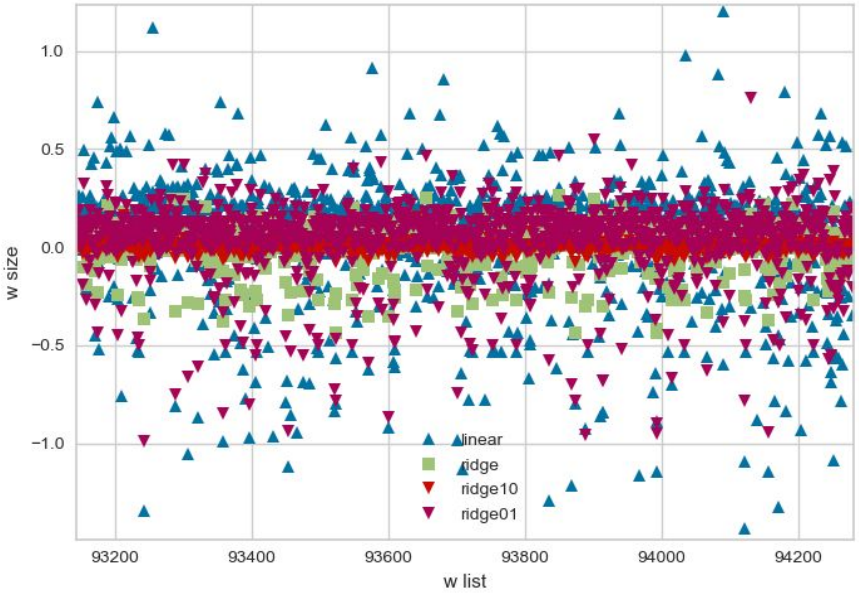
Linear Regression은 어떤 변수에 다른 변수들이 주는 영향력을 선형적으로 분석하는 가장 간단하고 대표적인 방법이다. 다른 통계방법에 비해 간단하나 해석력이 뛰어나 여전히 널리쓰이고 있고 다른 방법들의 기초가 되는 지식이다. 선형회귀는 변수값의 차이가 어디에서 비롯되는자 알고자 할 떄 사용하며, 독립변수(X)를 가지고 숫자형 종속변수(Y)를 가장 잘 설명, 예측 하는 선형관계를 찾는 방법이다.

	훈련 정확도	테스트 정확도
Linear	99.8%	40%
α = 0.1	98.8%	65.3%
α = 1	88.2%	65%
α = 10	65%	59%

위의 결과와 같이 train set으로 정확도를 평가했을 때와 test set으로 정확도를 평가했을 때 값의 차이가 크게 나는 것을 보고 과대적합이라 판단하여 규제를 해보기로 하였다. 여기에는 ridge 회귀를 사용했는데 릿지 회귀에서의 가중치(w) 선택은 훈련데이터를 잘 예측하기

위해서 뿐만 아니라 제약 조건을 만족시키기 위한 목적도 있다. 이 말은 가중치의 절댓값을 가능한 한 작게 만들어야 한다는 것이다.

위에서 실행한 결과처럼 규제를 통해 모델이 과대적합되지 않도록 제한한다. 아래의 사진은 규제(arpha)의 값에 따라 가중치의 절댓값을 나타낸 표이다. alpha가 10일 때 일반화과 된 모습은 볼 수 있지만 정확도를 봤을 때 우리가 기대했던 값까지는 못 미치기도 하고 너무 단순한 모델이라 판단하여 linear regression은 포기하기로 했다.



< 그림 12 > Linear Regression 일반화 결과

4-5. Decision Tree

이 프로젝트에서 malicious url을 찾아내는 것이 새 데이터가 들어왔을 때 기존에 가지고 있던 data set과 유사성을 얼마나 띄느냐로 판단하기 때문에 데이터들이 가진 속성들로부터 패턴을 찾아내서 분류하는 결정트리 모델도 적합할 것이라 생각했다.

	결정트리
훈련 데이터 정확도	100%
테스트 셋 정확도	91%

결정트리가 가진 단점인 과대적합이 결과에 나타났고, 이에 규제를 가해보았지만 성능이 테스트 셋 정확도가 70% 정도로 별로 좋지 않게 나와서 결정트리 모델 여러개를 사용하는 랜덤포레스트나 부스팅 기법을 사용해보기로 했다.

4-6. 앙상블(랜덤포레스트, 아다부스트, 그레디언트 부스트)

단일 모델 뿐만 아니라 여러 개의 모델들을 가지고 하나의 예측값을 만들어내는 앙상블 기법도 활용해보았다. 우선 여러개의 결정트리가 독립적인 결정을 내린 후 이를 종합하여 하나의 예측값을 만들어내는 랜덤포레스트 모델을 테스트 해봤다. 또한 여러개의 모델을 연결하여 이전모델의 오차로 부터 학습하는 아다부스팅 기법과 그래디언트 부스팅 기법도 함께 테스트했다. 아다부스팅 보다는 그레디언트 부스팅이 가중치를 부여할 때에 경사하강법을 통해 파라미터를 조정하기 때문에 더 성능이 좋을 것이라고 예상했다. 앙상블 기법은 훈련하는데에 시간이 너무 오래 걸려서 데이터셋의 사이즈를 줄여서 진행했다.

	훈련 데이터 정확도	테스트 셋 정확도
랜덤포레스트	99.7%	86.5%
아다부스트	88.9%	83.9%
그레디언트 부스트	89.4%	86.1%

결과는 예상대로 그레디언트 부스트가 아다부스트보다 성능이 좀 더 좋았고 데이터 클래스를 예측하는 데에 걸리는 시간도 조금 더 적었다. 다만 랜덤포레스트가 과대적합을 보여서 규제를 가하여 사용해보기로 하였고, 두 부스트 기법 중 비교적 더 좋은 성능을 보인 그레디언트 부스트도 함께 사용해보기로 하였다.

5. 최종 모델 선정

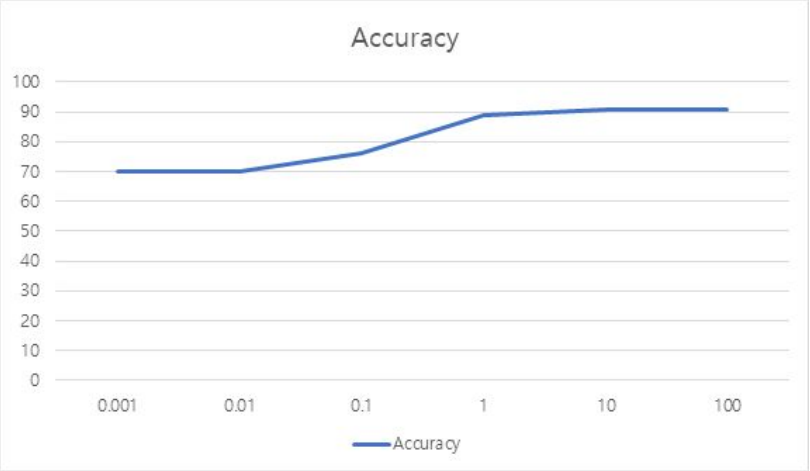
데이터의 전처리와 훈련시킬 모델을 선정했으니 남은건 일반화 성능을 높이는 것과 malicious url을 걸러내는 것이 프로젝트의 목적인 만큼 재현율보다 정밀도가 더 중요하기 때문에 정밀도를 높이는 데에 집중했다. 우선 각 모델의 규제는 그리드 탐색을 통해 최적의 파라미터 값을 찾아내어 성능평가를 해봤다.

5-1 로지스틱 회귀

로지스틱 회귀는 하이퍼 파라미터를 그리드 서치를 통해서 찾았다.
밑의 표와 그래프는 그리드 서치를 통해 찾은 파라미터와 정확도 등을 정리한 자료들이다.

C 값	test set 정확도	교차검증 평균	정밀도
100000	92.6%	90.4%	93.3%
10000	92.7%	90.4%	92.6%
1000	92.7%	90.4%	92.5%
100	92.7%	90.4%	92.5%
10	92.5%	90.4%	92.25%
1	91.4%	89.3%	90.9%
0.1	87.4%	85.1%	86.2%

그리드 탐색 결과 로지스틱회귀는 C 값이 클수록 즉, 규제를 줄일 때 가장 높은 성능을 보였다.



< 그림 13 > 하이퍼 파라미터에 따른 정확도

실제로는 C=10을 이용하였다. 위의 표와 그래프를 보면 알 수 있듯이 , C가 10이 된 이후로는 규제를 풀어도 성능이 크게 늘지 않는다는 것을 알 수 있다. 그렇기 때문에 C의 값은 10이 적정하다고 판단하였다.

5-2 랜덤포레스트

랜덤포레스트는 파라미터 n_estimator, max_depth, max_features로 진행했다.
n_estimator의 경우 숫자가 증가할수록 성능이 올라갔으나 200 이후부터는 증가폭이 작아져서 200이 최적이라고 판단했다.

max_features의 경우 결과는 다음과 같이 13일때 가장 높음을 알 수 있었다.

max_features	테스트셋 정확도
10	90.31%
11	90.11%
12	90.24
13	90.4
14	90%

max_depth는 값이 증가할수록 정확도가 높아지다가 16을 넘어가면서 부터는 다시 낮아지는 경향을 보였다.

5-3 그레디언트 부스트

그레디언트 부스트의 경우는 파라미터 learning rate, max_depth, n_estimators, min_samples_leaf, min_samples_split 을 가지고 그리드탐색을 진행했다.
밑은 각각의 파라미터에 대한 정리이다.

- learning rate : 0.5 일때 가장 높았고 그 이후부터는 점차 정확도가 낮아졌다.

- max_depth max_depth : 2부터 측정해봤을 때 max_depth가 9일때 성능이 가장 좋았고 9를 넘어서부터는 성능이 점점 저하되었다.

- min_samples_leaf : 10~20을 가질때 성능이 높았으며 그중 값이 14일 때 성능이 가장 좋았다.

- min_samples_split : 10~20에서 높은 성능을 보였으며 10에서 가장 성능이 좋았다.
  다음 표들은, 각각의 파라미터들을 통해서 정확도를 파악한 표들이다.

learning rate 값	테스트셋 정확도
0.2	87.75%
0.3	87.91%
0.4	87.91%
0.5	88.72%
0.6	88.62%

n_estimators	테스트셋 정확도
100	84.95%
70	84.3%
50	76.48%

min_samples_leaf 값	테스트셋 정확도
12	83.75%
13	83.65%
14	83.79%
15	83.58%

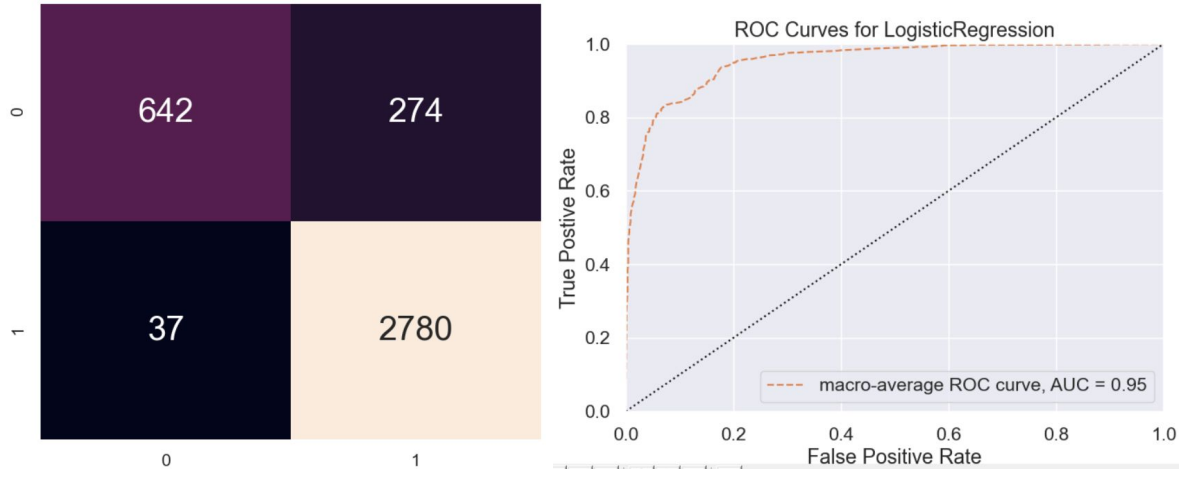
min_samples_split	테스트셋 정확도
10	86.35%
13	86.32%
7	86.32%
16	86.06%
20	86.03%

5-4 모델 정리

그리드 탐색을 통해 얻어낸 각 모델의 최적의 파라미터를 통해 규제를 가하여 Cmatrix와 ROC 커브를 그려서 성능평가를 진행해보았다.

- 로지스틱 회귀 : C = 10, solver = 'lbfgs', penalty = 'l2'
- 랜덤포레스트 : n_estimators = 200, max_features = 13, max_depth = 16
- 그레디언트 부스팅 : learning_rate = 0.5, max_depth = 8, n_estimators = 100, min_samples_leaf = 14, min_samples_split = 10

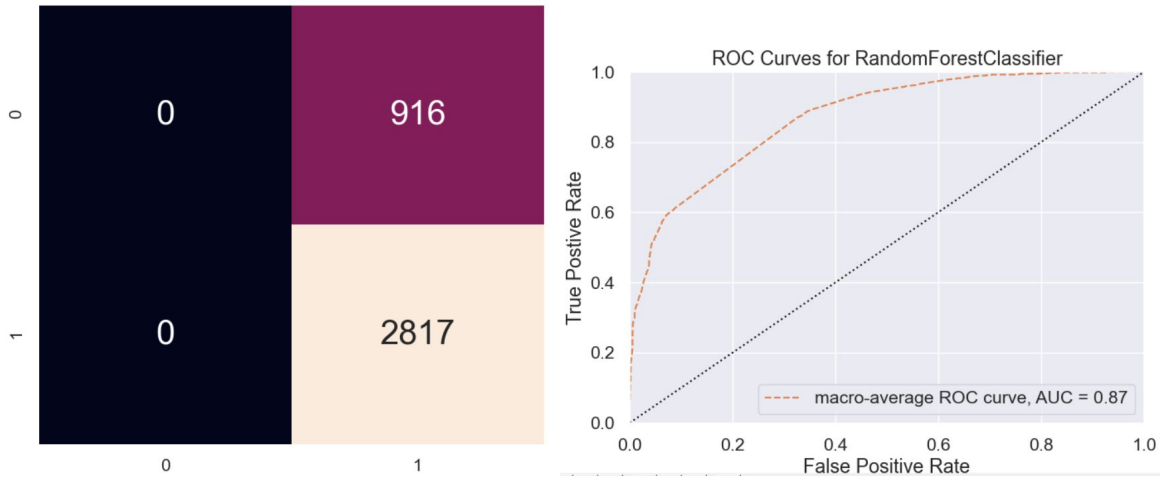
a) 로지스틱 회귀



< 그림 14 > 로지스틱 회귀 성능 평가

정밀도: 91.02%
재현율: 98.68%

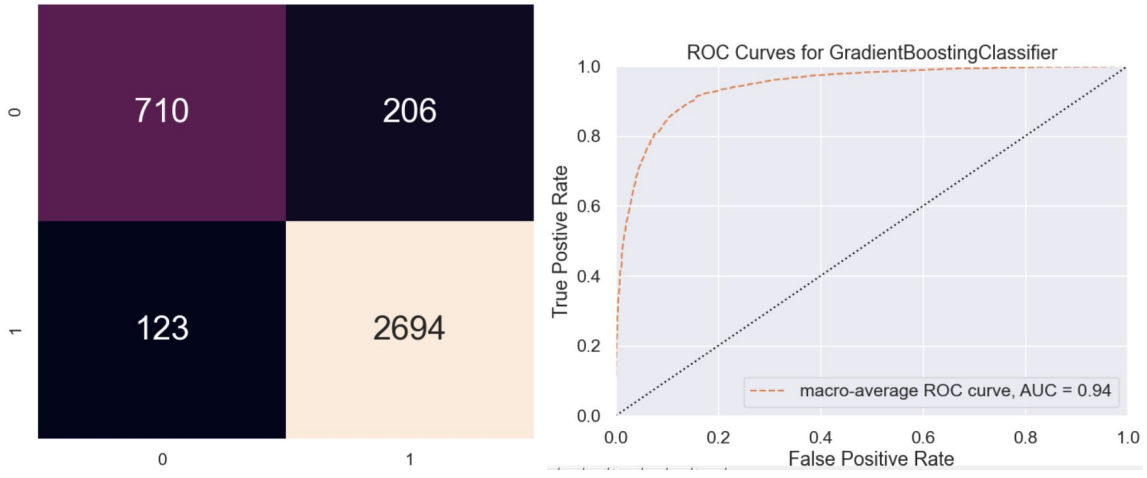
b) 랜덤포레스트



< 그림 15 > 랜덤 포레스트 성능 평가

정밀도 : 75.46%
재현율 : 100%

c) 그레디언트 부스팅



< 그림 16 > 그래디언트 부스팅 성능 평가

정밀도 : 92.89%
재현율 : 95.63%

```
pred = logit.predict(X_test)
scores = cross_val_score(logit, X, y, cv = 5)
print(scores)
pd.DataFrame(cross_validate(logit,X,y,cv=5))
print("일반화 성능 평균: ", scores.mean())

pred = rnd_clf.predict(X_test)
scores = cross_val_score(rnd_clf, X, y, cv = 5)
print(scores)
pd.DataFrame(cross_validate(rnd_clf,X,y,cv=5))
print("일반화 성능 평균: ", scores.mean())

pred = logit.predict(X_test)
scores = cross_val_score(grad, X, y, cv = 5)
print(scores)
pd.DataFrame(cross_validate(grad,X,y,cv=5))
print("일반화 성능 평균: ", scores.mean())
```

< 그림 17 > 교차검증 코드

일반화 성능 평가는 교차검증을 통해 이뤄졌다.

- 로지스틱 회귀 - 86.67%
- 랜덤포레스트 - 75.06%
- 그레디언트 부스트 - 86.53%

로지스틱회귀와 그레디언트 부스트는 일반화 성능이 많이 좋아졌으나 랜덤포레스트의 경우는 그리디 탐색을 통해 최적의 파라미터 값을 줬음에도 모든 데이터를 malicious url이 아니라고 예측하는 등 성능면에서 큰 하자를 보였다. 이를 통해 그리디 서치가 항상 정답은 아니라는 것을 깨달았다. 로지스틱 회귀 모델과 그레디언트 부스트 모델은 성능 면에서는 큰 차이를 보이지 않았다. 하지만 스팸메일을 걸러낼 때 FN보다 FP가 중요했던 것처럼 malicious url을 정상 url이라고 판단하지 않는 것이 중요하다고 생각해서 정밀도를 가장 우선시 하였다. 따라서 정밀도면에서 조금 더 우수한 성능을 보인 그레디언트 부스트를 우리 프로젝트의 최종 모델로 선정했다.

6. 최종 결과

위와 같은 과정을 통해서 우리는 알고 있는 지식들을 이용하여, 최종적인 결과물을 생성하였다.
먼저 전처리 과정에서는 공통 domain을 가진 URL을 삭제하고 tf-idf 벡터라이징을 이용한 전처리 과정을 거쳤다.
tf-idf 를 이용하기 위해서는 , 각 url을 단어 단위로 나누어야 했다. 밑의 코드 사진은, url을 단어 단위로 나누어서 벡터를 만드는 코드이다.

```
def makeTokens(f):
    tkns_BySlash = str(f.encode('utf-8')).split('/')
    total_Tokens = []
    tkns_BySlash = f.split("\n")
    for i in range(len(tkns_BySlash)):
        total_Tokens = total_Tokens+re.findall(r"[\w']+", tkns_BySlash[i])
    total_Tokens = list(set(total_Tokens))
    return total_Tokens
```

<그림 18> url을 단어 단위로 나누는 코드

```
vectorizer = TfidfVectorizer(tokenizer=makeTokens)
X = vectorizer.fit_transform(url_list)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = TEST_SIZE, random_state = RS)
```

<그림 19> 그림 18에서 나눈 단어를 tf-idf를 적용하는 코드

위의 과정을 거친 이후에는 미리 생성해 둔 url로부터 모델(그레디언트 부스트를 미리 학습)을 로딩하였다. 해당 과정을 통해서 학습 시간을 낮추어, 프로그램을 켜서 바로 이용할 수 있도록 하였다.

```
vectorizer = TfidfVectorizer(tokenizer=makeTokens)
X = vectorizer.fit_transform(url_list)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)


grad = GradientBoostingClassifier(learning_rate = 0.5, max_depth = 8
                                  , n_estimators = 100, min_samples_leaf = 14, min_samples_split = 10).fit(X_train, y_train)
joblib.dump(grad, 'grad.pkl')
```

```
vectorizer = pickle.load(open("vectorizer.pickle",'rb'))
model = joblib.load('grad.pkl')
```

<그림 20> 미리 학습을 시켜서, 모델을 pkl로 저장하는 코드와 해당 모델을 로드하는 과정

이 후, 셀레니엄을 이용하여, 크롬을 동작시켰다. 해당 드라이버를 통해서 사용자가 웹사이트를 이동하는 것을 감지할 수 있게 되었다. 만약 내가 웹사이트를 이동을 했다면 make_link_list함수를 호출하여 현재 url에서 하이퍼링크, 즉 a태그를 모두 가지고 와서 리스트 형식으로 반환을 하도록 하였다. 이 후, 이 리스트를 이용하여 예측하고 현재 링크 중에 bad가 있다면 이를 감지하고 tk 모듈을 통해서 알림을 띄워줬다.

밑의 <그림 21>중 윗 부분은 make_link_list 함수를 통해서 현재 url에 있는 a.href 즉 하이퍼링크를 모두 가지고 오는 부분이고, 밑의 코드는 해당 링크를 통해서 실제로 판별하고 알림을 띄워주는 코드이다.

```
def make_link_list(url):
    Dict={}
    req=urllib.request.Request(url, headers={'User-Agent': 'Mozilla/5.0'})
    html=urllib.request.urlopen(req)
    tmp=BeautifulSoup(html,"html.parser")

    for link in tmp.find_all('a'):
        tmp=link.get('href')
        value=link.text.strip()[0:10]
        if(len(value)>=10):  #열글자 넘을 경우
            value+="..."
        try:
            if('http' in tmp):
                Dict[value]=tmp  #Dict[tmp]=value
        except:
            continue
    return Dict.values()
```

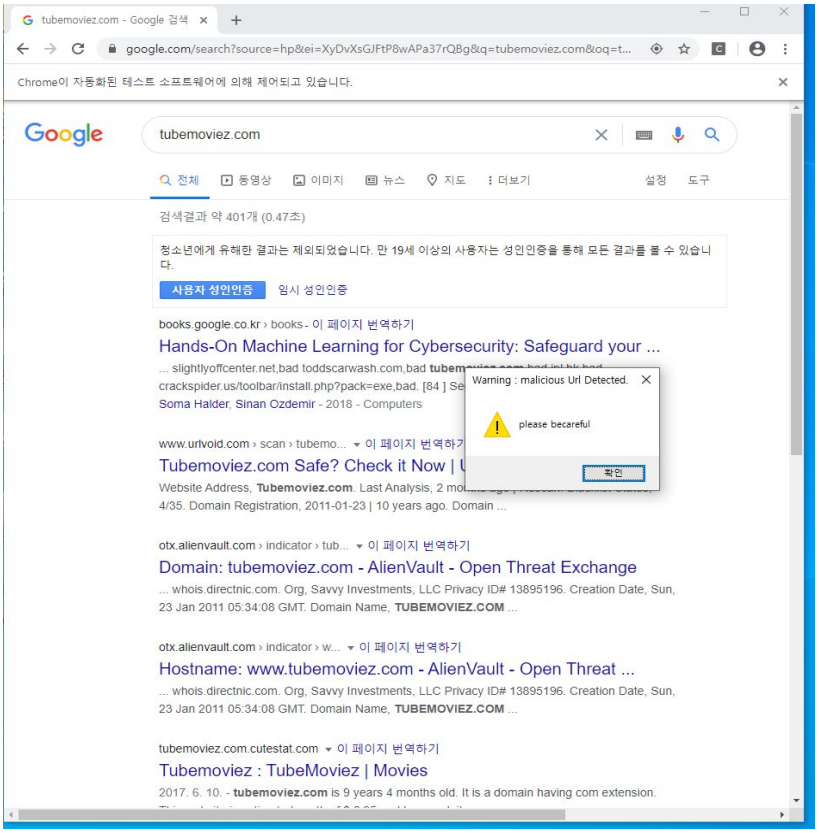
```
driver = webdriver.Chrome(executable_path=r'C:\\Users\\hojin\\Desktop\\chromedriver.exe')

root=Tk()
root.withdraw()

while 1:
    time.sleep(1)
    if(current_url != driver.current_url):
        current_url=driver.current_url
        url_list = make_link_list(current_url)
        X_pre = vectorizer.transform(url_list)
        New_pre = model.predict(X_pre)
        if 'bad' in New_pre:
            msg.showwarning('Warning : malicious Url Detected.')
```

<그림 21> 현재 url로부터 링크를 가지고오는 부분(상), 프로그램 주요 로직(하)

그리고 마지막으로 밑의 <그림22>는 실제로 해당 프로그램을 통해서 악성 url을 탐지하였을 때 경고 알람을 띄워주는 부분이다.



< 그림 22 > 악성 url 예측

아쉬운 점은 악성 사이트가 아닌 곳에서도 경고창이 뜬다는 것이었다. 이에 원인을 생각해봤는데 첫번째로는 데이터가 부족했다는 점이었고 두번째는 feature가 부족했다는 점이었다. url의 feature로 각 url의 reputation이나 도메인 만료 기간 등 여러가지를 추가하여 학습했다면 좀 더 좋은 성능이 나오지 않았을까하는 아쉬움이 남았다.