AI기반 악성코드 탐지 프로젝트

결과 보고서

과목명	정보보호와 시스템보안
교수님	윤명근 교수님
제출일	2020.11.27
팀명 (팀원)	18 [팀장] 정보보안암호수학과 20182219 박민진 정보보안암호수학과 20182212 김은주 정보보안암호수학과 20182236 윤혜진

AI를 이용한 악성코드 탐지 프로그램 구현

팀 명: 18

[팀장]정보보안암호수학과 20182219 박민진

[팀원]정보보안암호수학과 20182212 김은주

[팀원]정보보안암호수학과 20182236 윤혜진

1 서론

4차 산업혁명의 핵심기술 중 하나인 인공지능(AI)는 2016년 3월 구글의 알파고(AlphaGo)와 이세돌 9단의 바둑 대결 이후로 인공지능 인식이 매우 높아졌다. 이후 인공지능은 세계적으로 안전, 국방, 의료, 금융, 보안 등 다양한 분야에 적용되고 있다. 인공지능(Artificial Intelligence, AI)이란 인간이 가진 지각, 학습, 추론, 자연언어 처리 등의 능력을 컴퓨터가 실행할 수 있도록 프로그램으로 구현하는 기술로 기계학습(머신러닝), 딥러닝, 자연어 처리 등 첨단기술을 개발하는 방향으로 발전되고 있다. 최근 인공지능은 보안, 의료, 국방 분야에서 활용하기위해 연구가 활발히 이루어지고 있다.

보안뉴스의 한 기사에 따르면 올해 초부터 발생한 코로나 바이러스(COVID-19)의 장기화로 인해 비대면, 원격근무가 일상화되면서 기관·기업, 개인의 개인용 기기를 이용한 환경에서 해커와의 접점이 늘어나 더 많은 보안 이슈가 발생하고 있다고 한다.[i] 또한 디지서트에서 2021년 사이버 보안 동향을 예측한 자료를 보면 온라인 상에서의 보안이 중요해지고 '뉴 노멀'이 공격받을 것이라고 내다봤다.[ii] 뉴 노멀이란 시대 변화에 따라 새롭게 부상하는 표준을 의미한다. 앞서 두 보도자료에 따라 해커들이 사용자가 쉽게 접근가능한 앱, 메일, 클라우드 등을 피싱 공격으로 악성바이러스, 악성코드를 전파하는 등의 공격이 증가할 것으로 예상한다.

이 글에선 위에서 언급한 악성 코드를 이용한 공격으로부터 예방하기 위해 4차산업혁명의 핵심기술인 인공지능(AI)를 이용한 악성코드 탐지 프로그램 구현 결과를 설명하고자 한다.

2 프로그램 구성

AI기반 악성코드 탐지 프로그램은 크게 특징 추출, 학습데이터 구성, 검증 데이터 구성, 학습, 검증, 앙상블, 테스트 데이터 예측, 특징 중요도 판별로 나뉜다. 2에서는 각 구성에 대한 주어진 tutorial.ipynb와 다른 부분의 코드를 설명한다.

2.1 특징 추출

2.1.1 PEMINER

2.1.1.1 중요도 상위 40개의 정보로 특징으로 생성하는 함수

- 아래의 코드는 PEMINER의 정보 중 중요도가 높은 상위 40개를 선택하여 특징으로 생성하는 함수이다.
- 중요도를 판별하는 방법과 함수는 아래 2.5.1에서 설명한다.
- PE-miner에서 제공하는 189개의 특징을 모두 사용하는 것보다는 악성코드를 판별하는데 중요한 정보를 가지고 특징으로 선택하는 것이 정확도가 높아 tutorial.inpyb에서 수정하였다.
    - peminer_header_40: 중요도가 높은 40개의 헤더를 저장한 리스트

```
peminer_header_40 = ['OptionalHeader.ImageBase',
'OptionalHeader.DataDirectory.IMAGE_DIRECTORY_ENTRY_SECURITY.Size', …,
'OptionalHeader.DataDirectory.IMAGE_DIRECTORY_ENTRY_TLS.VirtualAddress']

def process_report(self):
    for _, value in sorted(self.report.items(), key=lambda x: x[0]):
       if _ in peminer_header_40:
           self.vector.append(value)
    return self.vector
```

2.1.2 EMBER

- 중요도 그래프를 통해 Ember의 정보 중에서 중요한 정보만 선택하여 특징으로 생성하였다.
- 중요도가 높은 정보만 선택해서 넣었을 경우 정확도가 떨어지는 경우에는 헤더의 전체 정보가 들어가도록 하였다.
- 중요도 판별하는 방법과 함수는 아래 2.5.2에서 설명한다.

2.1.2.1 Byte Entropy 값을 정규화 하는 함수

- Ember의 정보 중에서 byte entropy에 해당하는 정보를 불러와 byte entropy의 총 합으로 나누어 실수화(정규화)하는 함수이다.

```
def get_byteentropy_info(self):
```

```
    byteentropy = np.array(self.report["byteentropy"])
    total = byteentropy.sum()
    vector = byteentropy / total
    return vector.tolist()
```

· 아래의 함수는 중요도에 따라 특징을 추출하기 위해 구현했던 byte entropy 특징 추출 함수이다.

```
def get_byteentropy_info(self):
        byte_entropy = [0, 2, 5, 8, 9, 14, 17, 19, 20, 22, 24, 28, 29, 31, 32, 34, 35,
                        37, 38, 39, 41, 46, 47, 48, 50, 55, 56, 57, 58, 59, 61, 63, 64,
                        66, 67, 68, 69, 70, 72, 74, 75, 76, 79, 80, 81, 83, 85, 86, 87,
                        88, 89, 90, 91, 95, 96, 100, 101, 102, 103, 104, 105, 106, 107,
                        108, 109, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123,
                        125, 126, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138,
                        139, 140, 141, 143, 144, 146, 147, 148, 149, 150, 151, 152, 153,
                        154, 156, 157, 158, 159, 161, 162, 163, 165, 166, 167, 169, 170,
                        171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 182, 183, 184,
                        185, 186, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 200,
                        201, 202, 203, 204, 205, 207, 208, 209, 210, 211, 213, 214, 215,
                        216, 217, 218, 219, 220, 221, 222, 223, 226, 228, 229, 230, 231,
                        232, 233, 234, 235, 237, 238, 240, 242, 244, 245, 246, 249, 253,
                        254, 255]
        byteentropy1 = []
        byteentropy2 = np.array(self.report["byteentropy"])
        for i in byte_entropy:
          byteentropy1.append(byteentropy2[i])
        byteentropy = np.array(byteentropy1)
        total = byteentropy.sum()
        vector = byteentropy / total
        return vector.tolist()
```

2.1.2.2 Import 정보를 특징으로 추출하는 함수

- 사용한 DLL 이름을 실수 벡터 256차원으로 만든다.
- 함수 이름은 DLL 이름과 함수 이름을 묶어서 실수 벡터 1024차원으로 만든다.

```
def get_imports_info(self):
        imports = self.report["imports"]
        libraries = list(set([l.lower() for l in imports.keys()]))
        libraries_hashed
         = FeatureHasher(256, input_type="string").transform([libraries]).toarray()[0]
        import_
         = [lib.lower() + ':' + e for lib, elist in imports.items() for e in elist]
        imports_hashed
         = FeatureHasher(1024, input_type="string").transform([import_]).toarray()[0]
        vector = np.hstack([libraries_hashed, imports_hashed])
        return vector.tolist()
```

```
def get_imports_info(self):
        Imports_DLL = [8, 11, 20, 25, 30, 43, 51, 60, 81, 200, 212, 217, 219, 228, 231,
                       247]
        Imports_Function = [0, 1, 2, 10, 26, 47, 52, 55, 59, 69, 84, 87, 104, 108, 111,
                            113, 114, 139, 142, 152, 167, 172, 178, 195, 200, 205, 217,
                            232, 233, 235, 244, 245, 266, 274, 282, 283, 302, 306, 312,
                            317, 327, 333, 335, 340, 347, 349, 354, 358, 361, 362, 367,
                            370, 373, 393, 398, 402, 404, 409, 411, 440, 446, 451, 487,
                            490, 496, 500, 502, 514, 528, 534, 535, 551, 557, 566, 574,
                            576, 581, 587, 600, 607, 611, 619, 637, 639, 650, 654, 678,
                            681, 684, 687, 689, 697, 698, 722, 740, 750, 751, 752, 753,
                            755, 770, 789, 795, 796, 808, 821, 827, 830, 834, 838, 839,
                            861, 864, 872, 873, 886, 893, 894, 895, 896, 898, 901, 907,
                            916, 920, 935, 942, 959, 960, 963, 970, 976, 985, 989, 995,
                            1000, 1012]
        libraries_hashed_list = []
        imports_hashed_list = []
        imports = self.report["imports"]
        libraries = list(set([l.lower() for l in imports.keys()]))
        libraries_hashed = FeatureHasher(256,
input_type="string").transform([libraries]).toarray()[0]
        for i in Imports_DLL:
          libraries_hashed_list.append(libraries_hashed[i])
        import_ = [lib.lower() + ':' + e for lib, elist in imports.items() for e in
elist]
        imports_hashed = FeatureHasher(1024,
input_type="string").transform([import_]).toarray()[0]
        for j in Imports_Function:
          imports_hashed_list.append(imports_hashed[j])
        vector = np.hstack([libraries_hashed_list, imports_hashed_list])
        return vector.tolist()
```

2.1.2.3 Data directory 정보를 특징으로 추출하는 함수

- 정보들의 이름과 가상 크기(virtual_address)와 크기(size)를 실수 벡터 256차원으로 만든다.

```
def get_datadirectories_info(self):
        Data_Directory = [0, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 19, 20, 21,
                          24, 25, 27, 29]
        datadirectories = self.report["datadirectories"]
        name_order = ["EXPORT_TABLE", "IMPORT_TABLE", "RESOURCE_TABLE",
                      "EXCEPTION_TABLE", "CERTIFICATE_TABLE","BASE_RELOCATION_TABLE",
                      "DEBUG", "ARCHITECTURE", "GLOBAL_PTR", "TLS_TABLE",
                      "LOAD_CONFIG_TABLE","BOUND_IMPORT", "IAT",
                      "DELAY_IMPORT_DESCRIPTOR", "CLR_RUNTIME_HEADER"]
        vector = np.zeros(19, dtype=np.float32)
        cnt = 0
        for i in Data_Directory:
          if i < len(datadirectories):
              if i%2 == 0:
                vector[cnt] = datadirectories[i//2]["size"]
              if i%2 == 1:
                vector[cnt] = datadirectories[i//2]["virtual_address"]
          cnt += 1
        return vector.tolist()
```

- 아래의 함수는 중요도에 따라 특징을 추출하기 이전에 구현했던 datadirectories 특징 추출 함수이다.

```
def get_datadirectories_info(self):
        datadirectories = self.report["datadirectories"]
        name_order = ["EXPORT_TABLE", "IMPORT_TABLE",
                      "RESOURCE_TABLE", "EXCEPTION_TABLE", "CERTIFICATE_TABLE",
                      "BASE_RELOCATION_TABLE", "DEBUG", "ARCHITECTURE",
                      "GLOBAL_PTR", "TLS_TABLE", "LOAD_CONFIG_TABLE",
                      "BOUND_IMPORT", "IAT", "DELAY_IMPORT_DESCRIPTOR",
                      "CLR_RUNTIME_HEADER"]
        vector = np.zeros(2 * len(name_order), dtype=np.float32)
        for i in range(len(name_order)):
          if i < len(datadirectories):
            vector[2 * i] = datadirectories[i]["size"]
            vector[2 * i + 1] = datadirectories[i]["virtual_address"]
        return vector.tolist()
```

2.1.2.4 Section 정보를 특징으로 추출하는 함수

- 파일에 포함된 모든 섹션 크기와 가상 길이, 엔트로피, 특성들과 섹션 중 엔트리 포인트를 포함하는 섹션들을 실수 벡터 50차원으로 만든다.

- 일반적인 섹션 정보는 섹션 개수와 섹션 크기가 0인 섹션 개수, 섹션 이름이 없

는 섹션 개수, 섹션 중 MEM_READ(메모리 읽기), MEM_EXCUTE(메모리 실행)이 가능한 섹션 개수, 섹션 중 MEM_WRITE(메모리 쓰기)가 가능한 섹션 개수로 이루어져 있다.

```
def get_section_info(self):
    Section_General = [1, 3, 4]
    Section_Size = [1, 4, 5, 14, 22, 28, 32, 35, 41, 42, 43]
    Section_Entropy = [5, 14, 20, 28, 32, 35, 40, 41, 42, 43]
    Section_Virtual_Size = [0, 4, 5, 10, 14, 16, 27, 32, 34, 35, 38, 41, 42, 43]
    Section_Entry = [9, 17, 20, 39]
    Section_Characteristic = [13, 23, 37, 47]

    sections = self.report["section"]
    vector = [
        #0. len(sections),  # total number of sections
        # number of sections with nonzero size
        sum(1 for s in sections['sections'] if s['size'] == 0),
        # number of sections with an empty name
        #2. sum(1 for s in sections['sections'] if s['name'] == ""),
        # number of RX
        sum(1 for s in sections['sections'] if 'MEM_READ' in s['props']
            and 'MEM_EXECUTE' in s['props']),
        # number of W
        sum(1 for s in sections['sections'] if 'MEM_WRITE' in s['props'])]

    # gross characteristics of each section
    section_sizes = [(s['name'], s['size']) for s in sections['sections']]
    section_sizes_hashed
     = FeatureHasher(50, input_type="pair").transform([section_sizes]).toarray()[0] # 50
    section_sizes_hashed_new = []
    for value in Section_Size:
      section_sizes_hashed_new.append(section_sizes_hashed[value])

    section_entropy = [(s['name'], s['entropy']) for s in sections['sections']]
    section_entropy_hashed
     = FeatureHasher(50, input_type="pair").transform([section_entropy]).toarray()[0]
    section_entropy_hashed_new = []
    for value in Section_Entropy:
      section_entropy_hashed_new.append(section_entropy_hashed[value])

    section_vsize = [(s['name'], s['vsize']) for s in sections['sections']]
    section_vsize_hashed
     = FeatureHasher(50, input_type="pair").transform([section_vsize]).toarray()[0]
    section_vsize_hashed_new = []
    for value in Section_Virtual_Size:
```

```
      section_vsize_hashed_new.append(section_vsize_hashed[value])

    entry_name_hashed =
     FeatureHasher(50, input_type="string").transform([sections['entry']]).toarray()[0]
    entry_name_hashed_new = []
    for value in Section_Entry:
      entry_name_hashed_new.append(entry_name_hashed[value])

    characteristics = [p for s in sections['sections'] for p in s['props']
                       if s['name'] == sections['entry']]
    characteristics_hashed =
     FeatureHasher(50, input_type="string").transform([characteristics]).toarray()[0]
    characteristics_hashed_new = []
    for value in Section_Characteristic:
      characteristics_hashed_new.append(characteristics_hashed[value])

    hstack = np.hstack([
        vector, section_sizes_hashed_new, section_entropy_hashed_new,
        section_vsize_hashed_new, entry_name_hashed_new,
        characteristics_hashed_new])

    return hstack.tolist()
```

- 아래의 함수는 중요도에 따라 특징을 추출하기 이전에 구현했던 section 정보에서 특징 추출 함수이다.

```
def get_section_info(self):
        sections = self.report["section"]
        vector = [
            len(sections),  # total number of sections
            # number of sections with nonzero size
            sum(1 for s in sections['sections'] if s['size'] == 0),
            # number of sections with an empty name
            sum(1 for s in sections['sections'] if s['name'] == ""),
            # number of RX
            sum(1 for s in sections['sections'] if 'MEM_READ' in s['props'] and
'MEM_EXECUTE' in s['props']),
            # number of W
            sum(1 for s in sections['sections'] if 'MEM_WRITE' in s['props'])
        ]
        # gross characteristics of each section
        section_sizes = [(s['name'], s['size']) for s in sections['sections']]
        section_sizes_hashed
         = FeatureHasher(50, input_type="pair").transform([section_sizes]).toarray()[0]
        section_entropy = [(s['name'], s['entropy']) for s in sections['sections']]
```

```
        section_entropy_hashed
        = FeatureHasher(50, input_type="pair").transform([section_entropy]).toarray()[0]
        section_vsize = [(s['name'], s['vsize']) for s in sections['sections']]
        section_vsize_hashed
        = FeatureHasher(50, input_type="pair").transform([section_vsize]).toarray()[0]
        entry_name_hashed
        = FeatureHasher(50, input_type="string").transform([sections['entry']])
                                                                        .toarray()[0]
        characteristics = [p for s in sections['sections'] for p in s['props']
                                                    if s['name'] == sections['entry']]
        characteristics_hashed
        = FeatureHasher(50, input_type="string").transform([characteristics])
                                                                        .toarray()[0]
        return np.hstack([
            vector, section_sizes_hashed, section_entropy_hashed,
            section_vsize_hashed, entry_name_hashed,
            characteristics_hashed
        ]).tolist()
```

2.1.2.5 Export 정보를 특징으로 추출하는 함수

- 함수 이름을 실수 벡터 256차원으로 만든다.

```
def get_exports_info(self):
        exports_hashed
         = FeatureHasher(128, input_type="string").transform([self.report]).toarray()[0]
        return exports_hashed.tolist()
```

2.1.2.6 Header file 정보를 특징으로 추출하는 함수

- COFF Header와 Optional Header의 값을 가져온다.

```
Header_Info = [0, 11, 14, 15, 28, 29, 34, 51, 52, 53, 54, 55, 57, 59, 60]
        fh1 = FeatureHasher(10, input_type="string").transform([self.report['header']
              ['coff']['characteristics']]).toarray()[0]
        fh1_new = []
        for i in [0,3,4]:
          fh1_new.append(fh1[i])
        fh2 = FeatureHasher(10, input_type="string").transform([[self.report['header']
              ['optional']['subsystem']]]).toarray()[0]  # 21-30 -> 28, 29
        fh2_new = []
        for i in [7,8]:
          fh2_new.append(fh2[i])
        fh3 = FeatureHasher(10, input_type="string").transform([self.report['header']
              ['optional']['dll_characteristics']]).toarray()[0]  # 31-40 ->34
        fh3_new = []
```

```
        fh3_new.append(fh3[3])

        hstack = np.hstack([
            self.report['header']['coff']['timestamp'], # 0
            #FeatureHasher(10, input_type="string").transform([[self.report['header']
              ['coff']['machine']]]).toarray()[0],  # 1~10
            fh1_new, # 11-20 -> 11,14,15
            fh2_new,  # 21-30 -> 28, 29
            fh3_new,  # 31-40 ->34
            #FeatureHasher(10, input_type="string").transform([[self.report['header']
              ['optional']['magic']]]).toarray()[0],  # 41-50
            self.report['header']['optional']['major_image_version'], # 51
            self.report['header']['optional']['minor_image_version'], # 52
            self.report['header']['optional']['major_linker_version'],# 53
            self.report['header']['optional']['minor_linker_version'],# 54
            self.report['header']['optional']['major_operating_system_version'],# 55
            #self.report['header']['optional']['minor_operating_system_version'],# 56
            self.report['header']['optional']['major_subsystem_version'],# 57
            #self.report['header']['optional']['minor_subsystem_version'],# 58
            self.report['header']['optional']['sizeof_code'],# 59
            self.report['header']['optional']['sizeof_headers'],# 60
            #self.report['header']['optional']['sizeof_heap_commit'],# 61
        ])
        return hstack.tolist()
```

- 아래의 함수는 중요도에 따라 특징을 추출하기 이전에 구현했던 header_file 정보에서 특징 추출 함수이다.

```
def get_header_file_info(self):
      return np.hstack([
            self.report['header']['coff']['timestamp'],
            FeatureHasher(10, input_type="string")
                  .transform([[self.report['header']['coff']['machine']]]).toarray()[0],
            FeatureHasher(10, input_type="string")
            .transform([self.report['header']['coff']['characteristics']]).toarray()[0],
            FeatureHasher(10, input_type="string")
            .transform([[self.report['header']['optional']['subsystem']]]).toarray()[0],
            FeatureHasher(10, input_type="string")
   .transform([self.report['header']['optional']['dll_characteristics']]).toarray()[0],
            FeatureHasher(10, input_type="string")
                .transform([[self.report['header']['optional']['magic']]]).toarray()[0],
            self.report['header']['optional']['major_image_version'],
            self.report['header']['optional']['minor_image_version'],
            self.report['header']['optional']['major_linker_version'],
            self.report['header']['optional']['minor_linker_version'],
```

```
            self.report['header']['optional']['major_operating_system_version'],
            self.report['header']['optional']['minor_operating_system_version'],
            self.report['header']['optional']['major_subsystem_version'],
            self.report['header']['optional']['minor_subsystem_version'],
            self.report['header']['optional']['sizeof_code'],
            self.report['header']['optional']['sizeof_headers'],
            self.report['header']['optional']['sizeof_heap_commit'],
        ]).tolist()
```

2.1.2.7 general file 정보 중에서 중요도에 따라 특징 추출하는 함수

- 아래의 함수는 중요도에 따라 특징을 추출하여 프로그램을 실행해 보았으나 성능(정확도)가 이전보다 중요도에 따라 추출하지 않은 것보다 낮아 아래의 함수를 사용하지 않았다.

```
def get_general_file_info(self):
        general = self.report["general"]
        vector = [
            general['size'], general['vsize'], general['has_debug'],
            general['exports'], general['imports'],
            general['has_relocations'], general['has_resources'],
            general['has_signature'], general['has_tls'], general['symbols']
        ]
        return vector
```

2.1.2.8 string 정보 중에서 중요도에 따라 특징을 추출하는 함수

- 아래의 함수는 중요도에 따라 특징을 추출하여 프로그램을 실행해 보았으나 성능(정확도)가 이전보다 중요도에 따라 추출하지 않은 것보다 낮아 아래의 함수를 사용하지 않았다.

```
def get_string_info(self):
        #String_Info = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 17, 18, 19,
                        20, 21, 22, 23, 25, 28, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39,
                        40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 56,
                        57, 58, 59, 60, 62, 63, 64, 66, 68, 69, 70, 71, 72, 74, 75, 76,
                        77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
                        93, 94, 95, 96, 97, 99, 100, 101, 103]
        String_Info_printabledist = [5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                                     21, 22, 23, 25, 28, 29, 31, 32, 33, 34, 35, 36, 37,
                                     38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
                                     51, 53, 54, 55, 56, 57, 58, 59, 60, 62, 63, 64, 66,
                                     68, 69, 70, 71, 72, 74, 75, 76, 77, 78, 79, 80, 81,
                                     82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
                                     95, 96, 97, 99, 100]
```

```
        strings = self.report["strings"]
        hist_divisor = float(strings['printables']) if strings['printables'] > 0 else
                                                                                    1.0
        vector = [
            strings['MZ'],
            strings['avlength'],
            strings['entropy'],
            strings['numstrings'],
            strings['paths'], # 5부터가 안에 있는 원소 첫번째
            strings ['printables'],
            strings['urls']
        ]
        for i in String_Info_printabledist:
            if i < len(strings['printabledist']):
              vector.append(strings['printabledist'][i-5] / hist_divisor)
        return vector
```

2.1.3 PESTUDIO

2.1.3.1 Entropy 최대값을 특징으로 추출하는 함수

- Pestudio 정보에 entropy의 최대값을 구해 6.0이상이면 1, 아니면 0으로 수치화한 특징을 벡터로 생성하는 함수이다.
- entropy값이 6.0이상이면 파일이 암호화가 되어있거나 난독화되어있는 것으로 정상파일에서 흔히 볼 수 없는 정보라 특징으로 선택하였다.

```
def get_byteentropy_info(self):
    byteentropy = np.array(self.report["byteentropy"])
    total = byteentropy.sum()
    vector = byteentropy / total
    return vector.tolist()
```

2.1.3.2 Certificate 정보를 특징으로 추출하는 함수

- Pestudio 정보에 Certificate 정보가 있으면 0, 없으면 1으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_certificate_info(self):
      try:
            certificate = self.report["image"]["certificate"]

            if certificate == "n/a":
                result = 1
            else:
                result = 0
```

```
        except:
            result = -1
        return [result]
```

2.1.3.3 Debug 정보를 특징으로 추출하는 함수

- Pestudio 정보에 Debug 정보가 있으면 0, 없으면 1으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_debug_info(self):
        try:
            debug = self.report["image"]["debug"]
            if debug == "n/a":
                result = 1
            else:
                result = 0
        except:
            result = -1
        return [result]
```

2.1.3.4 File Header내의 network_run_from_swap 정보를 특징으로 추출하는 함수

- Pestudio 정보 중에서 file-header에 해당하는 network-run-from-swap 정보가 true 면 1, 아니면 0으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_network_run_from_swap_info(self):
        try:
            network_run_from_swap = self.report["image"]["file-header"]
                                                        ["network-run-from-swap"]
            if network_run_from_swap == "true":
                result = 1
            else:
                result = 0
        except:
            result = -1
        return [result]
```

2.1.3.5 Optional Header내의 control_flow_guard 정보를 특징으로 추출하는 함수

- Pestudio 정보 중에서 optional-header에 해당하는 control_flow_guard 정보가 있으면 0, 없으면 1으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_control_flow_guard_info(self):
```

```
        try:
            control_flow_guard = self.report["image"]["optional-header"]
                                                                ["control-flow-guard"]
            if control_flow_guard == "n/a":
                result = 1
            else:
                result = 0
        except:
            result = -1
        return [result]
```

2.1.3.6 Export 정보를 특징으로 추출하는 함수

- Pestudio 정보에 Export 정보가 있으면 0, 없으면 1으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_exports_info(self):
        try:
            exports = self.report["image"]["exports"]
            if exports == "n/a":
                result = 1
            else:
                result = 0
        except:
            result = -1
        return [result]
```

2.1.3.7 tls_callbaks 정보를 특징으로 추출하는 함수

- Pestudio 정보에 tls_callbacks 정보가 있으면 0, 없으면 1으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_tls_callbacks_info(self):
        try:
            tls_callbacks = self.report["image"]["tls-callbacks"]
            if tls_callbacks == "n/a":
                result = 1
            else:
                result = 0
        except:
            result = -1
        return [result]
```

2.1.3.8 Section 정보를 특징으로 추출하는 함수

- Pestudio 정보에서 Section에 해당하는 정보 중에서 section의 수, 최대 entropy값, 최소 entropy값, excutable 정보를 특징으로 추출하는 함수이다.

-

```
def get_section_info(self):
        try:
            sections = self.report["image"]["sections"]["section"]
            num_of_section = len(section)

            max_entropy = 0.0
            min_entropy = 50.0
            excutable = 0
            for section in sections:
                max_entropy = max(max_entropy, float(section["@entropy"]))
                min_entropy = min(min_entropy, float(section["@entropy"]))

                if section["@excutable"] == "-":
                    excutable += 1
                else:
                    excutable += 0
        except:
            num_of_section = 0
            max_entropy = -1.0
            min_entropy = -1.0
            excutable = -1
        return [num_of_section, max_entropy, min_entropy, excutable]
```

2.1.3.9 Import 정보를 특징으로 추출하는 함수

- Pestudio 정보에서 Import에 해당하는 정보 중에 함수의 이름에 해당하는 “@name”을 불러와 정상파일과 악성파일의 함수 상위 10개의 빈도수를 나타내는 리스트로 해당 파일의 함수가 포함되는지 아닌지를 특징으로 추출하는 함수이다.

- 정상파일과 악성파일의 함수의 상위 10개를 판별하는 함수는 아래 2.6에서 설명한다.

```
def get_import_info(self, flag = 0):
        dic_o = {'VerQueryValueA': 3826, 'GetFileVersionInfoA': 3381,
                 'GetFileVersionInfoSizeA': 3376, 'lstrcmpiA': 3359,
                 'HeapReAlloc': 3285, 'FindNextFileA': 3272,
                'SetErrorMode': 3241, 'DispatchMessageA': 3241, 'IsWindowEnabled': 3200}
        vector = []
        try:
            if flag == -1:
              raise Exception
            imports = self.report["image"]["imports"]["import"]
```

```
            for key,value in dic_o.items():
              # print(key)
              if key in imports["@name"]:
                vector.append(1)
              else:
                vector.append(0)
        except:
            vector = [-1 for i in range(len(dic_o))]
        return vector

def get_import_info2(self, flag = 0):
        dic_x = {'': 159559, 'RegQueryValueExA': 5448, 'RegOpenKeyExA': 5358,
                 'SelectObject': 5280, 'GetDeviceCaps': 5019, 'SetTextColor': 4967,
                 'CharNextA': 4890, 'SetBkColor': 4879, 'FindFirstFileA': 4766,
                 'SetBkMode': 4735, 'DestroyWindow': 4552, 'GetStartupInfoA': 4496,
                 'GetFileType': 4437, 'lstrcpynA': 4427, 'ImageList_Create': 4411}
        vector = []
        try:
            if flag == -1:
              raise Exception
            imports = self.report["image"]["imports"]["import"]
            for key,value in dic_x.items():
              # print(key)
              if key in imports["@name"]:
                vector.append(1)
              else:
                vector.append(0)
        except:
            vector = [-1 for i in range(len(dic_x))]
        return vector
```

2.1.3.10 virustotal 정보를 특징으로 추출하는 함수

- Pestudio 정보에서 virustotal에 해당하는 정보가 “offline”이면 1, 아니면 0으로 범주형 데이터를 수치화하여 특징으로 생성하는 함수이다.

```
def get_virustotal_info(self):
        try:
            if self.report["image"]["virustotal"] != "offline":
                result = 1
            else:
                result = 0
        except:
            result = -1
```

```
        return [result]
```

2.2 학습데이터 구성

2.2.1 전체 데이터 가져오기

- "PEMINER/학습데이터/"에 저장되어 있는 파일을 가져와 Queue에 저장하는 함수이다.
- Peminer와 Ember 파일에 저장되어 있는 학습데이터를 비교해보니 모든 데이터의 이름이 같으므로 Peminer의 하위 디렉토리에 있는 학습데이터를 가져온다.

```
import os
import queue

learn_data = []

def get_subdir(path):
    global  learn_data
    try:  # 검색이 허가되지 않은 디렉토리 접근에 관한 예외처리
        dirfiles = os.listdir(path)  # path에 해당하는 디렉토리 dirfiles에 추가
    except PermissionError:
        return []

    subdir_list = []
    for each in dirfiles:
        if each.endswith(".json"):  # .txt로 끝나는 파일은 따로 all_txt리스트에 저장
            learn_data.append(path + each)
        full_name = path + each  # path와 each(디렉토리)
        if os.path.isdir(full_name):
            subdir_list.append(full_name + "/")
    return subdir_list

dir_queue = queue.Queue()
# /Library/ 아래의 모든 하위 디렉토리 찾기
dir_queue.put("PEMINER/학습데이터/")  # 검색하고자 하는 디렉토리를 줄 앞에 세움

while not dir_queue.empty():  # 큐가 비어있는 상태인지 확인
    dir_name = dir_queue.get()  # 큐에서 빼낸 dir_name
    # dir_name의 하위 디렉토리를 subdir_names에 저장
    subdir_names = get_subdir(dir_name)
    for each in subdir_names:  # subdir_names리스트에 있는 디렉토리들을 다시 큐에 넣음
        dir_queue.put(each)
```

2.2.2 PESTUDIO 데이터 가져오기

- “PESTUDIO/학습데이터/”에 저장되어 있는 파일을 가져와 Queue에 저장하는 함수이다.
- Pestudio의 모든 학습데이터가 Peminer/Ember에 존재하지만 Pestudio에 존재하지 않는 데이터가 Peminer/Ember에 존재하기 때문에 따로 데이터를 가져온다.

```
import os
import queue

pestudio_data = []

def get_subdir(path):
    global pestudio_data
    try:  # 검색이 허가되지 않은 디렉토리 접근에 관한 예외처리
        dirfiles = os.listdir(path)  # path 에 해당하는 디렉토리 dirfiles 에 추가
    except PermissionError:
        return []

    subdir_list = []
    for each in dirfiles:
        if each.endswith(".json"):  # .txt 로 끝나는 파일은 따로 all_txt 리스트에 저장
            pestudio_data.append(path + each)
        full_name = path + each  # path 와 each(디렉토리)
        if os.path.isdir(full_name):
            subdir_list.append(full_name + "/")
    return subdir_list

dir_queue = queue.Queue()
# /Library/ 아래의 모든 하위 디렉토리 찾기
dir_queue.put("PESTUDIO/학습데이터/")  # 검색하고자 하는 디렉토리를 줄 앞에 세움

while not dir_queue.empty():  # 큐가 비어있는 상태인지 확인
    dir_name = dir_queue.get()  # 큐에서 빼낸 dir_name
    # dir_name 의 하위 디렉토리를 subdir_names 에 저장
    subdir_names = get_subdir(dir_name)
    for each in subdir_names:  # subdir_names 리스트에 있는 디렉토리들을 다시 큐에 넣음
        dir_queue.put(each)
```

2.2.3 학습데이터 특징 추출

- Peminer, Ember의 하위 디렉토리에 저장된 파일이름에 따라 Peminer, Ember의 하위 디렉토리에 저장된 .json 정보를 통해 특징을 추출하고 Pestudio의 경우 Peminer, Ember 하위 디렉토리에 저장된 파일이 없는 경우도 있어 파일이 있을 경우에는 Pestudio 정보에 따른 특징을 추출한다. 만약 파일이 Pestudio 하위 디렉토리에 저장되어 있지 않다면 특징 벡터를 -1로 채운다.

```
# 데이터의 특징 벡터 모음(2차원 리스트) : X
```

```
# 데이터의 레이블 모음(1차원 리스트) : y
X, y = [], []

for path in learn_data:
  feature_vector = []
  temp, fdata, fname = path.strip().split("/")
  fname = fname.replace(".json","")
  label = learn_label_table[fname]
  for data in ["PEMINER", "EMBER","PESTUDIO"]:
        path = f"{data}/{fdata}/{fname}.json"
        if data == "PEMINER":
            feature_vector += PeminerParser(path).process_report()
        elif data == "EMBER":
            feature_vector += EmberParser(path).process_report()
        elif data == "PESTUDIO" and path in pestudio_data:
            feature_vector += PestudioParser(path).process_report()
        else:
            feature_vector += PestudioParser(path, -1).process_report(-1)
  X.append(feature_vector)
  y.append(label)

np.asarray(X).shape, np.asarray(y).shape
```

2.3 검증데이터 구성

- 2.2 학습데이터 구성과 동일한 방법으로 코드를 구현하였다. 단, 파일은 검증데이터의 하위 디렉토리에 저장되어 있다.

2.3.1 전체 데이터 가져오기

```
import os
import queue

veri_data = []

def get_subdir(path):
    global veri_data
    try:  # 검색이 허가되지 않은 디렉토리 접근에 관한 예외처리
        dirfiles = os.listdir(path)  # path에 해당하는 디렉토리 dirfiles에 추가
    except PermissionError:
        return []

    subdir_list = []
    for each in dirfiles:
        if each.endswith(".json"):  # .txt로 끝나는 파일은 따로 all_txt리스트에 저장
            veri_data.append(path + each)
```

```
        full_name = path + each  # path와 each(디렉토리)
        if os.path.isdir(full_name):
            subdir_list.append(full_name + "/")
    return subdir_list

dir_queue = queue.Queue()
# /Library/ 아래의 모든 하위 디렉토리 찾기
dir_queue.put("PEMINER/검증데이터/")  # 검색하고자 하는 디렉토리를 줄 앞에 세움

while not dir_queue.empty():  # 큐가 비어있는 상태인지 확인
    dir_name = dir_queue.get()  # 큐에서 빼낸 dir_name
    # dir_name의 하위 디렉토리를 subdir_names에 저장
    subdir_names = get_subdir(dir_name)
    for each in subdir_names:  # subdir_names리스트에 있는 디렉토리들을 다시 큐에 넣음
        dir_queue.put(each)
```

2.3.2 PESTUDIO 데이터 가져오기

```
import os
import queue

pestudio_veri_data = []

def get_subdir(path):
    global pestudio_veri_data
    try:  # 검색이 허가되지 않은 디렉토리 접근에 관한 예외처리
        dirfiles = os.listdir(path)  # path에 해당하는 디렉토리 dirfiles에 추가
    except PermissionError:
        return []

    subdir_list = []
    for each in dirfiles:
        if each.endswith(".json"):  # .txt로 끝나는 파일은 따로 all_txt리스트에 저장
            pestudio_veri_data.append(path + each)
        full_name = path + each  # path와 each(디렉토리)
        if os.path.isdir(full_name):
            subdir_list.append(full_name + "/")
    return subdir_list

dir_queue = queue.Queue()
# /Library/ 아래의 모든 하위 디렉토리 찾기
dir_queue.put("PESTUDIO/검증데이터/")  # 검색하고자 하는 디렉토리를 줄 앞에 세움

while not dir_queue.empty():  # 큐가 비어있는 상태인지 확인
    dir_name = dir_queue.get()  # 큐에서 빼낸 dir_name
```

```
    # dir_name의 하위 디렉토리를 subdir_names에 저장
    subdir_names = get_subdir(dir_name)
    for each in subdir_names:  # subdir_names리스트에 있는 디렉토리들을 다시 큐에 넣음
        dir_queue.put(each)
```

2.3.3 검증데이터 특징 추출

```
# 데이터의 특징 벡터 모음(2차원 리스트) : X
# 데이터의 레이블 모음(1차원 리스트) : y
A, b = [], []

for path in veri_data:
  feature_vector = []
  temp, fdata, fname = path.strip().split("/")
  fname = fname.replace(".json","")
  label = verify_label_table[fname]
  for data in ["PEMINER", "EMBER","PESTUDIO"]:
        path = f"{data}/{fdata}/{fname}.json"
        if data == "PEMINER":
            feature_vector += PeminerParser(path).process_report()
        elif data == "EMBER":
            feature_vector += EmberParser(path).process_report()
        elif data == "PESTUDIO" and path in pestudio_veri_data:
            feature_vector += PestudioParser(path).process_report()
        else:
            feature_vector += PestudioParser(path, -1).process_report(-1)
  A.append(feature_vector)
  b.append(label)

np.asarray(A).shape, np.asarray(b).shape
```

2.4 테스트 데이터 구성 및 테스트

2.4.1 테스트 데이터 구성

- 학습시킨 모델로 테스트할 데이터들을 불러온다. 2.2 학습데이터 구성과 동일한 방법으로 코드를 구현하였다. 단, 파일은 검증데이터의 하위 디렉토리에 저장되어 있다.

2.4.1.1 전체 데이터 가져오기

```
import os
import queue

test_data = []
```

```
def get_subdir(path):
    global test_data
    try:  # 검색이 허가되지 않은 디렉토리 접근에 관한 예외처리
        dirfiles = os.listdir(path)  # path에 해당하는 디렉토리 dirfiles에 추가
    except PermissionError:
        return []

    subdir_list = []
    for each in dirfiles:
        if each.endswith(".json"):  # .txt로 끝나는 파일은 따로 all_txt리스트에 저장
            test_data.append(path + each)
        full_name = path + each  # path와 each(디렉토리)
        if os.path.isdir(full_name):
            subdir_list.append(full_name + "/")
    return subdir_list

dir_queue = queue.Queue()
# /Library/ 아래의 모든 하위 디렉토리 찾기
dir_queue.put("PEMINER/테스트데이터/")  # 검색하고자 하는 디렉토리를 줄 앞에 세움

while not dir_queue.empty():  # 큐가 비어있는 상태인지 확인
    dir_name = dir_queue.get()  # 큐에서 빼낸 dir_name
    # dir_name의 하위 디렉토리를 subdir_names에 저장
    subdir_names = get_subdir(dir_name)
    for each in subdir_names:  # subdir_names리스트에 있는 디렉토리들을 다시 큐에 넣음
        dir_queue.put(each)
```

2.4.1.2 Pestudio 테스트 데이터 가져오기

```
import os
import queue

pestudio_test_data = []

def get_subdir(path):
    global pestudio_test_data
    try:  # 검색이 허가되지 않은 디렉토리 접근에 관한 예외처리
        dirfiles = os.listdir(path)  # path에 해당하는 디렉토리 dirfiles에 추가
    except PermissionError:
        return []

    subdir_list = []
    for each in dirfiles:
```

```
        if each.endswith(".json"):  # .txt로 끝나는 파일은 따로 all_txt리스트에 저장
            pestudio_test_data.append(path + each)
        full_name = path + each  # path와 each(디렉토리)
        if os.path.isdir(full_name):
            subdir_list.append(full_name + "/")
    return subdir_list


dir_queue = queue.Queue()
# /Library/ 아래의 모든 하위 디렉토리 찾기
dir_queue.put("PESTUDIO/테스트데이터/")  # 검색하고자 하는 디렉토리를 줄 앞에 세움

while not dir_queue.empty():  # 큐가 비어있는 상태인지 확인
    dir_name = dir_queue.get()  # 큐에서 빼낸 dir_name
    # dir_name의 하위 디렉토리를 subdir_names에 저장
    subdir_names = get_subdir(dir_name)
    for each in subdir_names:  # subdir_names리스트에 있는 디렉토리들을 다시 큐에 넣음
        dir_queue.put(each)
```

2.4.1.3 테스트 데이터 특징 추출

```
# 데이터의 특징 벡터 모음(2차원 리스트) : T
T = []
file_name = []
for path in test_data:
  global file_name
  feature_vector = []
  temp, fdata, fname = path.strip().split("/")
  fname = fname.replace(".json","")
  file_name.append(fname)
  for data in ["PEMINER", "EMBER", "PESTUDIO"]:
        path = f"{data}/{fdata}/{fname}.json"
        if data == "PEMINER":
            feature_vector += PeminerParser(path).process_report()
        elif data == "EMBER":
            feature_vector += EmberParser(path).process_report()
        elif data == "PESTUDIO" and path in pestudio_veri_data:
            feature_vector += PestudioParser(path).process_report()
        else:
            feature_vector += PestudioParser(path, -1).process_report(-1)
  T.append(feature_vector)

np.asarray(T).shape
```

2.4.2 테스트 실행 및 csv 파일 생성

```
def test(X, models):
    predicts = []
    for model in models:
        prob = [result for _, result in model.predict_proba(X)]
        predicts.append(prob)

    predict = np.mean(predicts, axis=0)
    predict = [1 if x >= 0.5 else 0 for x in predict]

    return predict

test_ret = test(T, models)

import csv

f = open('predict.csv', 'w', newline='')
wr = csv.writer(f)
wr.writerow(['file', 'predict'])

for i in range(len(test_ret)):
  wr.writerow([file_name[i], test_ret[i]])

f.close()
```

2.5 특징 중요도 판별

2.5.1 PEMINER

- feat_importances는 Peminer의 특징들의 중요도이므로 출력 후 내림차순으로 정리하여 중요도가 높은 header 일부분만 가져와 사용하기로 하였다.

- 내림차순으로 정리하기 전, 값이 0인 것도 있어서 0이 아닌 것들만 feat_importances_nozero 리스트에 저장하였다. 그 개수는 107개였다.

- feat_importances_nozero를 완성한 뒤 내림차순으로 정리하여 중요도가 높은 상위 40개, 50개, 60개, … 등을 테스트해본 결과 상위 40개의 header만 사용했을 때 정확도가 가장 높게 나와 40개의 header만 선택하였다.

```
def draw_feature_importance_plot():
    plt.figure(figsize=(10, 10))
    feat_importances = pd.Series(clf.feature_importances_, index=peminer_header)
    print(feat_importances.sort_values())     # 값 출력
    pd.set_option('display.max_column', 500)  # 항목 전체 출력하려고
    feat_importances.nlargest(24).plot(kind='bar')

    feat_importances_nozero = {}
```

```
    for i in range(0,188):
      if feat_importances.values[i] != 0:
        feat_importances_nozero[feat_importances.index[i]] = feat_importances.values[i]
    feat = []
    for i in range(0, 40):
      feat.append(sorted(feat_importances_nozero.items(),
                                          key=lambda x: x[1], reverse=True)[i][0])
    print(feat)

    plt.ylabel("score", fontsize=16)
    plt.yticks(fontsize=16)
    plt.xticks(fontsize=16)
    plt.savefig("peminer importance")

draw_feature_importance_plot()
```

	rf 정확도	lgb 정확도	앙상블
Peminer 특징 10개	0.9476	0.9539	0.954
Peminer 특징 20개	0.9452	0.9532	0.9543
Peminer 특징 30개	0.9476	0.9524	0.954
Peminer 특징 40개	0.9482	0.9538	0.9561
Peminer 특징 50개	0.9477	0.9536	0.9541
Peminer 특징 60개	0.9481	0.954	0.9532
Peminer 특징 70개	0.9479	0.9539	0.9545
Peminer 특징 80개	0.9471	0.9539	0.9557
Peminer 특징 90개	0.9478	0.9541	0.9553
Peminer 특징 100개	0.9481	0.9541	0.9553
Peminer 특징 107개	0.9478	0.9541	0.955
Peminer 특징 110개	0.948	0.9541	0.9553
Peminer 특징 120개	0.9486	0.9541	0.9559
Peminer 특징 130개	0.9477	0.9535	0.9558
Peminer 특징 140개	0.948	0.9535	0.9556
Peminer 특징 150개	0.9497	0.9535	0.9556
Peminer 특징 160개	0.947	0.9535	0.9547
Peminer 특징 170개	0.9488	0.9535	0.9558
Peminer 특징 180개	0.9462	0.9535	0.9547
Peminer 특징 188개	0.9488	0.9535	0.9558

2.5.2 EMBER

- Peminer와 마찬가지로, feat_importances를 출력하여 상위 몇 개만 가져와 사용하려고 하였으나, header 하나가 높은 중요도를 갖는 것이 아니라 header 속 특정 번째가 높은 중요도를 가져서 Peminer와 같은 방법으로는 하지 못했다.

- 대신 출력했을 때, 값이 0인 것만 제외해도 739개로, 처음 2031개보다 훨씬 많이 줄기 때문에 한 header의 모든 데이터가 들어가는 것이 아니라 0이 아닌 데이터만 골라서 넣기로 하였다.

- 예를 들어, 아래 코드 속 예시 Header Info는 62개의 데이터를 가지지만, 위와 같은 과정으로 걸렀을 때 15개의 데이터만 중요도를 가지므로 이 데이터들만 가져오는 필터링 과정을 EMBER 코드에 적용시켰다.

```
section, dll, function, eat = 50, 256, 1024, 128

header = [f"Byte Histogram {i}" for i in range(256)] +\
[f"Byte Entropy Histogram {i}" for i in range(256)] +\
[f"String Info {i}" for i in range(104)]+\
[f"General File Info {i}" for i in range(10)]+\
[f"Header Info {i}" for i in range(62)]+\
[f"Section(General) {i}" for i in range(5)]+\
[f"Section(Size) {i}" for i in range(section)]+\
[f"Section(Entropy) {i}" for i in range(section)]+\
[f"Section(Virtual Size) {i}" for i in range(section)]+\
[f"Section(Entry) {i}" for i in range(section)]+\
[f"Section(Characteristic) {i}" for i in range(section)]+\
[f"Imports(DLL) {i}" for i in range(dll)]+\
[f"Imports(Function) {i}" for i in range(function)]+\
[f"Exports(Function) {i}" for i in range(eat)]+\
[f"Data Directory {i}" for i in range(30)]



plt.figure(figsize=(10, 10))
feat_importances = pd.Series(clf.feature_importances_, index=header)
feat_importances.nlargest(24).plot(kind='bar')
print(feat_importances.sort_values())     # 값 출력
pd.set_option('display.max_row', 2391)  # 항목 전체 출력하려고
feat_importances_nozero = {}
for i in range(0,2381):
  if feat_importances.values[i] != 0:
    feat_importances_nozero[feat_importances.index[i]] = feat_importances.values[i]
feat = []
for i in range(0, 40):
  feat.append(sorted(feat_importances_nozero.items(), key=lambda x: x[1],
reverse=True)[i][0])
print(feat)
# 예시: Header_Info = [0, 11, 14, 15, 28, 29, 34, 51, 52, 53, 54, 55, 57, 59, 60]
plt.ylabel("score", fontsize=16)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
```

	rf 정확도	lgb 정확도	앙상블
수정 전(원래 코드)	0.9473	0.9542	0.9539
Header_Info 수정 후	0.9467	0.9533	0.9549

Section_Info	0.9489	0.9527	0.9544
Import_Info	0.9479	0.9518	0.9534
Datadirectories_Info	0.9497	0.9546	0.9548
General_File_info	0.9501	0.9521	0.9549
String_Info	0.9490	0.9541	0.9535
Byteentropy_Info	0.9492	0.9533	0.9532
Header_Info + General_File_Info	0.9491	0.9521	0.9542
Header_Info + General_File_Info + Datadirectories_Info+ Section_Info	0.9471	0.9538	0.9557

- 아래의 표는 Ember 정보 중에서 추출한 특징들을 제외하면서 나온 성능을 측정한 값 중 일부를 나타낸 것이다.

제외한 특징	rf 정확도	lgb 정확도	앙상블
General_File_info	0.9468	0.9534	0.9539
Histogram_Info	0.941	0.9526	0.9555
Byteentropy_Info	0.9459	0.9533	0.9544
String_Info	0.9463	0.9531	0.9542
Import_Info	0.945	0.9515	0.952
Datadirextories_Info	0.9477	0.9536	0.9551
Section_Info	0.9458	0.9518	0.9528
Export_Info	0.9488	0.9538	0.9553
Header_Info	0.9465	0.9517	0.9537
General_File_info + Histogram_Info	0.9495	0.9545	0.9548
General_File_info + Byteentropy_Info	0.948	0.9525	0.9539
General_File_info + String_Info	0.9485	0.9533	0.9538
General_File_info + Import_Info	0.9417	0.9499	0.9503
General_File_info + Datadirectories_Info	0.9461	0.9534	0.9523
General_File_info + Section_Info	0.9472	0.9511	0.9512
General_File_info + Export_Info	0.9471	0.9534	0.9537
General_File_info + Header_Info	0.9464	0.9546	0.9542
Histogram_Info + Byteentropy_Info	0.9497	0.9535	0.9559
General_File_info + Histogram_Info + Byteentropy_Info	0.9507	0.9533	0.9567
General_File_info + Histogram_Info + Byteentropy_Info + String_Info	0.9523	0.9559	0.9588
General_File_info + Histogram_Info + Byteentropy_Info	0.9508	0.9545	0.9559

+ Datadirextories_Info			
Histogram_Info + String_Info + Datadirextories_Info + Export_Info	0.9498	0.9551	0.9577
Histogram_Info + String_Info + Export_Info	0.9512	0.9541	0.957
General_File_info + Histogram_Info + Byteentropy_Info + Import_Info	0.9189	0.948	0.9517
General_File_info + Histogram_Info + Byteentropy_Info + Datadirextories_Info	0.9497	0.9545	0.9555
General_File_info + Histogram_Info + Byteentropy_Info + Section_Info	0.9496	0.9505	0.9528
General_File_info + Histogram_Info + Byteentropy_Info + Export_Info	0.9495	0.9533	0.9563
General_File_info + Histogram_Info + Byteentropy_Info + Header_Info	0.9504	0.9542	0.9553
Histogram_Info + Byteentropy_Info + String_Info	0.9547	0.9563	0.9595
Histogram_Info + String_Info	0.9509	0.9541	0.957

2.6 Pestudio Import 악성파일 함수 분류

- 주어진 모든 데이터에 대해 함수이름을 리스트에 저장하였다
- 정상파일, 악성파일에 대해 따로 저장하여 각각 상위 50개 중 안 겹치는 상위 14개, 15개를 추출하여 이를 각각 특징으로 선택하였다.

```
func_name = {}
for path in pestudio_data:   # pestudio_data, pestudio_veri_data 2가지 경우에 대해 실행
  temp, fdata, fname = path.strip().split("/")
  fname = fname.replace(".json","")
  path = f"PESTUDIO/{fdata}/{fname}.json"
  data = read_json(path)
  label = learn_label_table[fname]
  if "image" not in data:
    continue
```

```
  if "imports" not in data["image"]:
    continue
  if "import" not in data["image"]["imports"]:
    continue
  for pe_import in data["image"]["imports"]["import"]:
    if "@name" not in pe_import:
      continue
    if pe_import == "@name":
      continue
    name = pe_import["@name"]
    if name not in func_name:
      func_name[name] = 1
    else:
      func_name[name] += 1
```

- 다음은 정상파일, 악성파일에서 빈도수가 가장 높은 14개, 15개의 함수명이다. 이때 겹치는 것은 제외하였다.

```
dic_o = {'QueryPerformanceCounter': 3434, 'GetCurrentProcessId': 3415,
         'GetSystemTimeAsFileTime': 3246, 'TerminateProcess': 3227,
         'SetUnhandledExceptionFilter': 3074, 'InitializeCriticalSection': 2413,
         'LocalFree': 2399, '_initterm': 2394, 'GetDC': 2394, 'CoTaskMemFree': 2325,
         'CoCreateInstance': 2308, 'free': 2281, 'GetWindowRect': 2262,
         'TlsGetValue': 2260}

dic_x = {'': 159559, 'RegQueryValueExA': 5448, 'RegOpenKeyExA': 5358,
         'SelectObject': 5280, 'GetDeviceCaps': 5019, 'SetTextColor': 4967,
         'CharNextA': 4890, 'SetBkColor': 4879, 'FindFirstFileA': 4766,
         'SetBkMode': 4735, 'DestroyWindow': 4552, 'GetStartupInfoA': 4496,
         'GetFileType': 4437, 'lstrcpynA': 4427, 'ImageList_Create': 4411}
```

	rf 정확도	lgb 정확도	앙상블
pestudio 정상, 악성 20개 중 안 겹치는 경우 - 6,7개	0.9471	0.9538	0.9551
pestudio 정상, 악성 30개 중 안 겹치는 경우 - 10,10개	0.9479	0.9538	0.956
pestudio 정상, 악성 40개 중 안 겹치는 경우 - 12,12개	0.9474	0.9538	0.955
pestudio 정상, 악성 50개 중 안 겹치는 경우 - 14,15개	0.9464	0.9538	0.9559
pestudio 정상, 악성 60개 중 안 겹치는 경우 - 15,15개	0.947	0.9538	0.9555
pestudio 정상, 악성 70개 중 안 겹치는 경우 - 15,15개	0.947	0.9538	0.9555
pestudio 정상, 악성 100개 중 안 겹치는 경우 - 15,15개	0.947	0.9538	0.9555
pestudio 정상, 악성 200개 중 안 겹치는 경우 - 15,15개	0.947	0.9538	0.9555

3 결론 및 느낀 점

테스트 데이터 중 어떤 것이 악성인지 정상인지를 정답이 없어 프로젝트를 통해 구현한 프로그램이 어느 정도의 정확성으로 악성 파일을 판별하는 지 알 수는 없다. 하지만 학습데이터로 학습한 뒤 검증 데이터로 앙상블을 돌려본 결과 정확도가 0.9595, 즉 95.95%나왔다. 이를 통해 테스트 데이터를 2에서 설명한 함수들을 통해 추출한 특징벡터로 악성 파일인지 아닌지를 판별하면 95.95% 확률로 올바르게 예측할 수 있다.

이 프로젝트를 하면서 알게 된 점은 첫 번째로 특징이 많다고 좋은 게 아니었다. Peminer의 경우, 중요도가 0이 아닌 것들만 넣어 앙상블 했을 때 제일 높게 나올 거라 기대하였고 돌려본 결과 특징을 40개만 넣었을 때가 가장 높게 나왔지만, 중요도가 0인 것도 추가해보았을 때 정확도가 올라간 것을 확인할 수 있었다. 우리는 중요도가 0이 아닌 것들만 골라서 했을 때에 특징의 개수가 적어서 0인 것을 넣었을 때에도 중요도가 올라간 것이라고 추측했다. 다음 Ember의 경우, 중요도에 따라 특징을 추출하는 것을 목표로 하였으나 2.5.2의 코드의 결과를 보면 정보의 카테고리(Byte Histogram, String Info 등)에 해당하는 인덱스 값이 무작위로 나와서 이것을 중요도가 높은 상위 몇 개만 가져와 적용하기에는 특징을 추출하기 어려웠다. 그래서 중요도가 0인 것을 제외하고 코드를 다시 만들었으나 정확도가 떨어져서 각 카테고리 별로 중요도를 따르지 않은, 즉 중요도가 0인 것을 포함하는 코드를 사용하여 성능을 비교해보니 더 높아짐을 알 수 있었다.

두 번째로 어떤 것을 기준으로 하는 지에 따라서 결과가 달라진다는 것이었다. Pestudio의 경우, Import 함수에서 범주형 데이터 함수 이름을 어떻게 처리하면 좋을지를 고민해봤다. 주어진 정답을 사용하여 정상 파일과 악성 파일에서 사용되는 함수의 빈도를 가지고 특징을 추출해보는 방법을 다음과 같이 고안해 보았다.

(1) 악성 파일에서 사용되는 함수의 빈도를 가지고 특징 추출

(2) 전체 파일과 정상 파일에서 사용되는 함수의 빈도를 가지고 특징 추출

(3) 전체 파일과 악성 파일에서 사용되는 함수의 빈도를 가지고 특징 추출

(4) 정상 파일과 악성 파일에서 중복되는 것을 제외하고 악성 파일에서 사용되는 함수의 빈도를 가지고 특징 추출

- 주어진 데이터가 사용하는 함수 중에서 정상 파일과 악성 파일에 중복되는 함수를 제외한 악성 파일에서 사용되는 함수로 비교해서 있으면 1, 없으면 0으로 수치화 하였다.

(5) 정상 파일과 악성 파일에서 사용되는 함수의 빈도를 비교 후 특징 추출(겹치는 건 제외)

- 주어진 데이터가 사용하는 함수 중에서 정상 파일에서 사용되는 함수가 많은 지 악성 파일에서 사용되는 함수가 많은 지 비교해서 악성이 많으면 1, 아니면 0으로 수치화 하였다.

다섯 가지 방법을 고안해보고 직접 테스트 해본 결과, 5번의 방법이 가장 적절했다고 생각하여 선택하게 되었다.

세 번째로 모델 역시 많이 쓴다고 좋은 건 아니라는 것이다. 특징 추출 함수를 완성하기 전, 각 모델별로 학습을 돌려본 결과 다음과 같은 정확도를 얻었다.

모델명	정확도
Rf	0.9467
Lgb	0.9537
Lr	0.8264
Mlp	0.8498

Adaboost	0.8997
Dt	0.9088
Svm	0.8267
knn	0.8943

위와 같은 정확도를 가지고 앙상블을 하였을 때에, rf와 lgb 이외에 다른 모델을 넣으면 정확도가 떨어지는 것을 확인할 수 있었다.

앞서 서론에서 말했듯이, 보안, 의료, 국방 분야에서 적용되고 있는 AI를 이용해 악성코드 탐지 프로그램을 만들어 보니, 다수의 데이터로 학습한 뒤에는 AI가 스스로 탐지할 수 있게 된다는 점에서 왜 AI가 4차산업혁명의 핵심기술인지를 알게 되었다. 이번 프로그램은 머신러닝만 사용하여 프로그램을 만든 결과 95.9% 정도의 정확도를 가지는 탐지 프로그램을 만들었는데, 딥러닝을 사용하면 정확도가 더 높아질 것이라 기대한다.

참고문헌

[i] "뉴 노멀 시대, 모든 직원의 보안인식 제고가 필요하다." *보안뉴스*. 2020년 11월 20일 수정, 2020년 11월 22일 접속, https://www.boannews.com/media/view.asp?idx=92662&kind=3&search=title&find=%C4%DA%B7%CE%B3%AA19.

[ii] "디지서트, '2021년 사이버보안 전망' 발표." *ITWORLD*. 2020년 11월 17일 수정, 2020년 11월 22일 접속, https://www.itworld.co.kr/news/172249.