빅데이터 프로젝트 보고서

1. 데이터 수집 / 크롤링

- 국가동물보호정보시스템 Open API를 이용해 전국 유기·입양 동물 데이터를 월·지역 단위로 자동 수집
- 전국동물보호센터 정보표준데이터(Open API)를 통해 보호소 주소·연락처·위치(위도·경도) 정보 수집
- 농림축산검역본부 동물등록 현황 Open API를 이용해 연도·지역·출생연도별 등록두수 데이터 수집
- API 오류 대비 재시도 로직, 페이징 처리, 월 단위 청크 분할 등 안정적인 대용량 수집 구조 구현

initCollections.js

> initCollections.js는 MongoDB 'animals' 데이터베이스에서 사용할 주요 컬렉션(abandonments, shelters, registrations)을 초기 설정하고, 각 컬렉션에 필요한 인덱스를 생성하는 스크립트이다. 유기동물·보호소·등록현황 데이터가 대량으로 저장되기 때문에, 조회 성능과 데이터 정확성을 위해 고유 ID 인덱스, 날짜 인덱스, 지역 인덱스, 공간(2dsphere) 인덱스 등을 사전에 설정하여 이후 수집·분석 단계가 안정적으로 동작할 수 있도록 준비하는 역할을 한다.

```
import { MongoClient } from "mongodb";
```

→ MongoDB에 접속하고 쿼리하기 위해 공식 MongoDB 드라이버에서 MongoClient 클래스를 가져온다.

```
import dotenv from "dotenv";
dotenv.config();
```

→ .env 파일에 있는 환경변수(MONGODB_URI 등)를 읽어와 process.env로 사용할 수 있게 설정한다.

```
const uri = process.env.MONGODB_URI;
const client = new MongoClient(uri);
```

→ .env에서 MongoDB 접속 URI를 읽어와 클라이언트 객체를 만든다.

→ 아직 이 시점에는 실제로 연결되지는 않고, 나중에 connect()를 호출하면 DB에 접속된다.

```
async function run() {
```

→ MongoDB에 연결하고 컬렉션 인덱스를 생성하는 메인 함수이다.

→ 비동기 작업(접속, 인덱스 생성)을 처리하기 위해 async로 선언했다.

```
try {
  await client.connect();
```

```
    const db = client.db("animals");
```

→ try 블록에서 실제 DB에 접속한다.

→ client.connect()로 MongoDB 서버에 접속하고, "animals"라는 이름의 데이터베이스를 선택한다.

→ 이 DB 안에 유기동물, 보호소, 등록현황 컬렉션이 모두 들어간다.

```
console.log("🐾 MongoDB 연결 성공 — 컬렉션 및 인덱스 생성 시작");
```

→ DB 연결이 성공했음을 콘솔에 찍고, 이제 컬렉션과 인덱스를 만들겠다는 로그를 남긴다.

```
const ab = db.collection("abandonments");
```

→ 유기동물 데이터를 저장할 abandonments 컬렉션 핸들을 가져온다.

→ 컬렉션이 아직 없다면 MongoDB가 자동으로 생성한다.

```
await ab.createIndex({ uid: 1 }, { unique: true });
```

→ abandonments 컬렉션의 uid 필드에 대해 오름차순 인덱스를 만들고, unique 옵션을 걸어 같은 uid가 중복 삽입되지 않도록 한다.

→ 즉, 한 유기동물 개체가 여러 번 수집되더라도 마지막 데이터로만 업데이트되게 만드는 기본 키 역할이다.

```
await ab.createIndex({ eventDate: 1 });
```

→ 유기 발생일(eventDate) 기준으로 빠르게 정렬·검색할 수 있도록 인덱스를 만든다.

→ 연도별, 기간별 통계를 뽑을 때 성능을 올려준다.

```
await ab.createIndex({ "location.geo": "2dsphere" });
```

→ location.geo 필드를 2dsphere 인덱스로 지정하여, 지도 좌표(위도·경도) 기반 공간 검색이 가능하게 한다.

→ 예를 들어 “이 위치 반경 10km 안의 유기동물” 같은 공간쿼리를 할 수 있다.

```
await ab.createIndex({ "location.sido": 1 });
await ab.createIndex({ "location.sigungu": 1 });
```

→ 시도(sido), 시군구(sigungu)에 각각 단일 인덱스를 걸어 지역별 조회 속도를 높인다.

→ “인천광역시 전체 유기동물”, “인천광역시 부평구 유기동물” 같은 쿼리용.

```
await ab.createIndex({ species: 1 });
```

→ 종(개, 고양이, 기타)을 나타내는 species 컬럼에 인덱스를 만들어, 종별 통계·필터링 속도를 향상시킨다.

```
await ab.createIndex({ eventDate: 1, species: 1 });
```

→ eventDate + species 복합 인덱스를 만들어, “특정 기간 동안 특정 종의 유기동물” 같은 조건을 빠르게 검색할 수 있도록 최적화한다.

```
const sh = db.collection("shelters");
```

→ 보호소 정보가 저장될 shelters 컬렉션 핸들을 가져온다.

```
await sh.createIndex({ uid: 1 }, { unique: true });
```

→ shelters 컬렉션에서도 uid를 보호소 고유 ID로 사용하고, unique 인덱스를 걸어 중복 저장을 막는다.

```
await sh.createIndex({ "location.geo": "2dsphere" });
```

→ 보호소 위치(location.geo) 역시 공간 인덱스를 생성해 지도 기반 분석과 거리 계산을 할 수 있게 한다.

```
const rg = db.collection("registrations");
```

→ 반려동물 등록현황 데이터를 저장하는 registrations 컬렉션 핸들을 가져온다.

```
await rg.createIndex({ uid: 1 }, { unique: true });
```

→ registrations에서도 uid를 복합 키(시도+시군구+출생연도+종+품종 등)로 사용하고, unique 인덱스를 걸어 같은 집계 행이 중복으로 쌓이지 않도록 한다.

```
await rg.createIndex({ year: 1, month: 1, sigungu: 1, species: 1 });
```

→ 등록 데이터를 연도(year), 월(month), 시군구(sigungu), 종(species) 기준으로 자주 조회할 것을 고려해 복합 인덱스를 생성한다.

→ “2024년 3월 인천광역시 부평구 개 등록두수” 같은 쿼리를 빠르게 실행할 수 있다.

```
console.log("✅ 컬렉션 & 인덱스 생성 완료");
```

→ 전체 컬렉션에 필요한 인덱스 설정이 끝났음을 로그로 표시한다.

```
} catch (err) {
  console.error("❌ 오류 발생:", err.message);
```

→ 위 과정에서 에러가 나면 catch 블록으로 들어와, 어떤 오류가 났는지 메시지를 출력한다.

```
} finally {
  await client.close();
```

→ 성공이든 실패든 마지막에는 항상 finally가 실행되고, 여기서 DB 연결(client)을 닫아준다.

```
run();
```

→ 앞에서 정의한 run() 함수를 실제로 실행하여,

→ 1) MongoDB에 연결하고,

→ 2) animals DB의 abandonments, shelters, registrations 컬렉션에 필요한 인덱스를 한 번에 세팅해주는 초기화 작업을 수행한다.

2. 데이터 저장 / 추출

- 수집한 데이터를 MongoDB에 abandonments / shelters / registrations 컬렉션으로 구분 저장
- Open API 응답을 종·품종·성별·중성화·지역·좌표 등이 표준화된 스키마로 정규화하여 저장
- unique 인덱스와 2dsphere 공간 인덱스를 기반으로 중복 없는 빠른 조회 구조 구성

upsert_abandonments_korea.js

> upsert_abandonments_korea.js는 공공데이터포털 유기동물 API(v2)를 이용하여 전국 시도·시군구·월 단위로 유기동물 데이터를 수집하는 스크립트이다. 페이지 단위로 데이터를 안정적으로 요청하며, API 응답을 정규화한 뒤 MongoDB의 abandonments 컬렉션에 uid 기준으로 upsert하여 중복 없이 최신 데이터만 유지한다. 대용량 데이터를 다루기 위해 재시도 로직, 월별 수집 구조, 시군구 유무에 따른 처리 등을 구현한 전국 단위 배치 수집 프로그램이다.

```
import dotenv from "dotenv";
dotenv.config();
```

→ .env 파일에 저장된 환경 변수를 로딩한다.

→ 이후 process.env.MONGODB_URI, process.env.NAAS_KEY 사용 가능.

```
import { MongoClient } from "mongodb";
```

→ MongoDB에 연결하고 쿼리를 수행할 수 있는 공식 드라이버를 불러온다.

```
const { MONGODB_URI, NAAS_KEY } = process.env;
```

→ 데이터베이스 접근 정보(MONGODB_URI)와 공공데이터 API Key(NAAS_KEY)를 환경변수에서 가져온다.
→ API 호출 및 DB 저장에 필수적인 설정.

```
const BASE = "https://apis.data.go.kr/1543061/abandonmentPublicService_v2";
```

→ 유기동물 정보 API(v2)의 기본 URL.
→ 나중에 call("abandonmentPublic_v2") 형태로 endpoint만 붙여 사용한다.

```
const sleep = (ms) ⇒ new Promise((res) ⇒ setTimeout(res, ms));
```

→ fetch 실패 시 재시도할 때 사용하는 대기 함수.

```
const arrify = (x) ⇒ (Array.isArray(x) ? x : x ? [x] : []);
```

→ API 응답이 객체 1개일 수도, 배열일 수도 있어 항상 배열 형태로 맞춰주는 헬퍼.

```
function parseYmd(str) { ... }
```

→ "YYYYMMDD" 문자열을 JS Date 객체로 변환한다.
→ 잘못된 날짜는 즉시 에러를 던져 수집 중 오류를 빠르게 발견하게 한다.

```
function formatYmd(date) { ... }
```

→ Date 객체를 "YYYYMMDD" 문자열로 되돌린다.

```
function endOfMonth(date) { ... }
```

→ 해당 월의 마지막 날짜(말일)를 계산한다.

```
function buildMonthlyRanges(start, end) { ... }
```

→ 전체 기간(예: 20240101~20241231)을 "월 단위 청크"로 쪼갠다.

→ API는 큰 기간 요청 시 데이터 누락이 심하므로 월 단위로 안정적으로 요청해야 한다.

→ 반환 값 예:

[{ start:"20240101", end:"20240131", label:"2024-01" }, ...]

```
async function call(endpoint, params = {}, maxRetries = 5) { ... }
```

→ 공공데이터포털 API 호출을 담당하는 핵심 함수.

→ 기능 요약:

- BASE + endpoint + params → 최종 URL 생성
- fetch 요청 후 HTTP 상태코드 검사
- 500대(서버 오류)는 재시도, 400대(클라이언트 오류)는 즉시 실패
- JSON 파싱 실패 시 에러
- 응답 구조에서 response.body.items.item 을 추출
- 결과가 객체 1개일 경우도 arrify로 배열로 변환
  → 안정적인 데이터 수집을 위해 재시도 로직이 매우 중요하다.

```
async function fetchSidos() { ... }
```

→ sido_v2 API 호출로 전국 시도 목록을 1번만 가져온다.

→ 각 시도의 upr_cd(시도 코드), orgdownNm(시도 이름) 포함.

```
async function fetchSigungu(upr_cd) { ... }
```

→ 특정 시도 코드(upr_cd)에 대한 시군구 목록을 sigungu_v2 API에서 가져온다.

→ 세종특별자치시처럼 시군구가 없는 경우도 있으므로 후처리가 필요하다.

```
async function fetchAbandonments(params) { ... }
```

→ abandonmentPublic_v2 엔드포인트로 실제 유기동물 데이터를 받아오는 함수.

→ params: bgnde(시작일), endde(종료일), upr_cd(시도), org_cd(시군구), pageNo 등.

```
function normalizeAbandonment(r) { ... }
```

→ 공공데이터 API 응답 한 건을 “프로젝트 내부 표준 구조”로 변환한다.

→ 주요 처리:

- r.happenDt, noticeSdt, noticeEdt → Date 객체 변환
- kindCd "[개] 믹스견" → species="개", kindName="믹스견"
- sexCd M/F/Q → Male/Female/Unknown
- neuterYn Y/N/U → Neutered/Not Neutered/Unknown
- weight "6.8(Kg)" → 숫자 6.8 추출
- happenYear, happenMonth 생성
- raw: r 로 원본 전체 기록 저장 (디버깅/추후 파생변수 가능)

→ 이 normalize 과정 덕분에 CSV로 전처리할 때 훨씬 안정적으로 처리 가능해진다.

```
const client = new MongoClient(MONGODB_URI);
await client.connect();
const db = client.db("animals");
const col = db.collection("abandonments");
```

→ MongoDB animals DB의 abandonments 컬렉션을 준비한다.

→ 모든 수집 데이터는 여기로 upsert된다.

```
const AB_START = "20240101";
const AB_END   = "20241231";
```

→ 2024년 1년치 데이터를 수집하도록 날짜 범위를 설정한다.

```
const ranges = buildMonthlyRanges(AB_START, AB_END);
```

→ API 요청을 한 달 단위 블록 리스트로 변환한다.

```
const sidos = await fetchSidos();
```

→ 전국 시도 목록을 불러와 반복할 준비.

```
for (const sido of sidos) { ... }
```

→ 모든 시도(서울, 인천, 부산 등)를 순차 처리.

```
sigunguList = await fetchSigungu(upr_cd);
```

→ 각 시도의 시군구(부평구, 중구 등)를 조회한다.

→ 만약 없다면(세종 등) 시군구 단위 수집 대신 "시도 단독" 방식으로 호출한다.

```
for (const range of ranges) { ... }
```

→ 월 단위 수집 루프.

→ 예: 2024-01 → 2024-02 → ... → 2024-12

```
if (!sigunguList.length) {
    while(true) {
        const params = { bgnde, endde, upr_cd, pageNo };
        const items = await fetchAbandonments(params);
        if (!items.length) break;
```

→ 세종특별자치시처럼 시군구가 없으면 upr_cd만 붙여 API를 호출한다.

→ pageNo를 증가시키며 모든 데이터를 수집한다.

```
for (const sg of sigunguList) {
    while(true) {
        const params = { bgnde, endde, upr_cd, org_cd, pageNo };
        const items = await fetchAbandonments(params);
        if (!items.length) break;
```

→ 일반 시/도는 upr_cd + org_cd 모두 필요하다.

→ pageNo를 증가시키며 전체 페이지를 수집한다.

```
const docs = items.map(normalizeAbandonment);
```

→ 응답 레코드를 normalizeAbandonment()로 정규화한다.

```
const bulkOps = docs.map((doc) => ({
    updateOne: {
        filter: { uid: doc.uid },
        update: { $set: doc },
        upsert: true
```

```
    }
}));
```

→ uid 기준으로 중복을 막는 upsert 작업을 구성한다.

→ bulkWrite로 한번에 처리할 수 있도록 배열로 만든다.

```
await col.bulkWrite(bulkOps, { ordered:false });
```

→ abandonments 컬렉션에 대량 upsert 실행.

→ ordered:false → 중간에 일부 실패하더라도 전체 작업은 계속 진행.

```
console.log(🏁 전국 유기동물 수집 완료 — 총 upsert=${total});
```

→ 프로그램 실행이 정상적으로 끝났음을 출력.

→ 총 몇 건의 데이터가 upsert되었는지 표시.

```
await client.close();
```

→ MongoDB 연결을 닫고 종료.

```
main().catch(...);
```

→ main 실행 중 발생한 모든 오류를 잡아 로그로 표시하고 종료한다.

upsert_registrations_korea.js

> upsert_registrations_korea.js는 농림축산식품부 동물등록 현황 API를 호출하여 전국 시도별·출생연도별·RFID 여부별·축종별 등록 집계 데이터를 수집한다. 시도/시군구/출생연도/축종/품종 조합을 uid 값으로 생성하고, 이를 기준으로 MongoDB registrations 컬렉션에 upsert해 항상 최신 등록 현황만 유지한다. 이 데이터는 유기동물 데이터와 결합하여 '등록 대비 유기 비율' 분석 등에 활용된다.

```
import dotenv from "dotenv";
dotenv.config();
```

→ .env 파일의 환경변수를 현재 프로세스에 로드한다.

→ 이후 process.env.MONGODB_URI, process.env.MAFRA_KEY 등을 사용할 수 있다.

```
import { MongoClient } from "mongodb";
```

→ MongoDB 서버에 접속할 수 있도록 공식 드라이버에서 MongoClient를 불러온다.

```
const { MONGODB_URI, MAFRA_KEY } = process.env;
```

→ 환경변수에서 MongoDB 접속 URI와 농림축산부(MAFRA) OpenAPI 키를 꺼낸다.

```
if (!MONGODB_URI) throw new Error("MONGODB_URI 없음");
if (!MAFRA_KEY) throw new Error("MAFRA_KEY 없음");
```

→ 필수 환경변수가 없으면 바로 프로그램을 중단시켜, 잘못된 설정 상태에서 실행되지 않도록 한다.

```
const BASE = "http://211.237.50.150:7080/openapi";
const GRID = "Grid_20210806000000000612_1";
```

→ 동물등록 현황 OpenAPI의 기본 URL(BASE)과

→ 응답 JSON 내부의 루트 객체 이름(GRID, 즉 Grid_20210806000000000612_1)을 상수로 정의한다.

```
const CTPV_LIST = [
   "서울특별시",
   "부산광역시",
   ...
   "제주특별자치도",
];
```

→ 시도 이름(CTPV 파라미터 값)에 해당하는 전국 17개 시도명을 배열로 정의한다.

→ 이 배열을 돌면서 시도별로 데이터를 수집할 것이다.

```
function buildUrl(start, end, ctpv) {
   return `${BASE}/${MAFRA_KEY}/json/${GRID}/${start}/${end}
   ?CTPV=${encodeURIComponent(ctpv)}`;
}
```

→ 특정 시도(ctpv)에 대해,

→ START~END 구간의 데이터를 가져올 때 사용할 최종 URL을 만드는 함수.

→ API 형식: /{API_KEY}/json/Grid_.../{START}/{END}?CTPV=시도명

```
async function fetchPage(start, end, ctpv) {
    const url = buildUrl(start, end, ctpv);
    const res = await fetch(url);
    if (!res.ok) { ... 에러 처리 ... }

    const text = await res.text();
    const json = JSON.parse(text);
    const grid = json[GRID];

    if (!grid) throw new Error("응답에 GRID 객체 없음");
    const { totalCnt, result, row } = grid;

    if (!result || result.code !== "INFO-000") {
        throw new Error(`API 오류: code=${result?.code}, msg=${result?.message}`);
    }
    return { totalCnt, rows: row || [] };
}
```

→ 하나의 페이지 범위(START~END)에 대해 동물등록 현황을 실제로 호출하는 함수.

→ HTTP 에러 처리, JSON 파싱, 응답 구조 검증을 수행하고,

→ totalCnt(전체 건수)와 row(실제 데이터 배열)를 반환한다.

```
function normalizeRow(row) {
    const ctpv = row.CTPV || null;      // 시도명
    const sgg = row.SGG || null;        // 시군구명
    const brdt = row.BRDT || "";        // 생년 (YYYYMMDD 또는 YYYY 형태)
    const birthYear = brdt ? Number(String(brdt).slice(0, 4)) : null;
    const rfidType = row.RFID_SE || null;     // RFID 여부 구분
    const species = row.LVSTCK_KND || null;   // 축종 (개, 고양이 등)
    const kind = row.SPCS || null;            // 품종
    const cnt = row.CNT ? Number(row.CNT) : 0; // 해당 조합의 등록 마릿수
    const uid = [
        ctpv || "",
        sgg || "",
        birthYear || "",
        rfidType || "",
        species || "",
        kind || "",
        ].join("|");

    return {
        uid,
        sido: ctpv,
        sigungu: sgg,
        birthYear,
```

```
        rfidType,
        species,
        kind,
        count: cnt,
        raw: row,
        };
    }
```

→ API에서 내려주는 한 행(row)을 프로젝트 내부에서 쓰기 편한 형태로 변환한다.

→ CTPV, SGG, BRDT, RFID_SE, LVSTCK_KND, SPCS, CNT 필드를 sido/sigungu/birthYear/rfidType/species/kind/count 로 맵핑하고,

→ 시도/시군구/출생연도/RFID/축종/품종을 붙여 uid라는 고유 키를 만든다.

→ raw에는 원본 row 전체를 그대로 저장해 디버깅과 추후 파생변수 계산에 쓸 수 있다.

```
async function main() {
    const client = new MongoClient(MONGODB_URI);
    await client.connect();
    const db = client.db("animals");
    const col = db.collection("registrations");
    console.log("✅ MongoDB 연결 — 전국 동물등록 현황 수집 시작");
```

→ MongoDB animals DB에 연결하고 registrations 컬렉션을 선택한다.

→ 이 컬렉션에 등록현황 집계 데이터를 upsert할 것이다.

```
let totalGlobal = 0;
```

→ 전국 모든 시도에 대해 upsert된 총 건수를 누적할 변수.

```
for (const ctpv of CTPV_LIST) {
    console.log(`\n==================== [${ctpv}] 수집 시작 ====================`);
    let start = 1;
    const pageSize = 1000;
    let firstTotalCnt = null;
    let localTotal = 0;
```

→ 한 시도(ctpv)에 대해

- start: 이번 요청의 시작 인덱스
- pageSize: 한 번에 가져올 행 수(1000)
- firstTotalCnt: 첫 페이지에서 확인한 전체 행 수
- localTotal: 이 시도에서 실제 upsert된 건수를 관리하기 위한 변수들이다.

```
while (true) {
const end = start + pageSize - 1;
  console.log(
    `📄 CTPV=${ctpv} start=${start} end=${end} 페이지 요청 중...`
  );

  const { totalCnt, rows } = await fetchPage(start, end, ctpv);
```

→ 이 시도에서 이번 구간(start~end)에 해당하는 페이지를 API로 호출한다.

→ totalCnt: 이 시도에 존재하는 전체 레코드 수

→ rows: 이번 페이지 범위 안에 포함된 실제 데이터 행 배열

```
  if (firstTotalCnt === null) {
    firstTotalCnt = totalCnt;
    console.log(`➡️ [${ctpv}] totalCnt=${totalCnt}`);
  }
```

→ 이 시도를 처음 호출하는 경우에만 totalCnt를 따로 기록해둔다(참고용).

→ 이후 로그에서 "추정 totalCnt"처럼 출력한다.

```
  if (!rows.length) {
    console.log(
      `⚠️ registrations 호출 중 오류 (CTPV=${ctpv}, start=${start}, end=${end},
      시도 1/5): 응답에 ${GRID} 필드가 없습니다.`
    );
    break;
  }
```

→ rows가 비어 있으면 더 이상 가져올 데이터가 없거나 응답 구조에 문제가 있는 것으로 보고 while 루프를 종료한다.

```
  const docs = rows.map(normalizeRow);
```

→ 이번 페이지에서 가져온 원시 데이터(rows)를 모두 normalizeRow로 정규화한다.

```
  const bulkOps = docs.map((doc) => ({
    updateOne: {
      filter: { uid: doc.uid },
      update: { $set: doc },
      upsert: true,
```

```
    },
  }));
```

→ 각 정규화 문서에 대해 uid 기준으로 updateOne + upsert:true 연산을 구성한다.

→ 이미 같은 uid 행이 있으면 덮어쓰고, 없으면 새로 삽입하여 항상 "최신 집계 상태"만 유지되도록 한다.

```
  if (bulkOps.length) {
    const res = await col.bulkWrite(bulkOps, { ordered: false });

    const upserted =
      (res.insertedCount || 0) +
      (res.upsertedCount || 0) +
      (res.modifiedCount || 0);

    localTotal += upserted;
    totalGlobal += upserted;

    console.log(
      `✅ [${ctpv}] start=${start} end=${end} upsert=${upserted}
      (시도 누적=${localTotal}, 전체 누적=${totalGlobal})`
    );
  }
```

→ bulkWrite로 registrations 컬렉션에 대량 upsert를 실행한다.

→ res.insertedCount, upsertedCount, modifiedCount를 합쳐 이번 페이지에서 영향을 받은 문서 수를 계산하고, 시도별(localTotal)·전체(totalGlobal) 누적 카운터를 갱신한다.

```
  if (end >= totalCnt) {
    console.log(
      `ℹ️ [${ctpv}] end=${end} >= totalCnt=${totalCnt} → 마지막 페이지로 판단`
    );
    break;
  }

  if (rows.length < pageSize) {
    console.log(
      `ℹ️ [${ctpv}] rows.length=${rows.length} < pageSize=${pageSize}
      → 마지막 페이지로 판단`
    );
    break;
  }
```

→ 두 가지 조건 중 하나라도 만족하면 "마지막 페이지"라고 판단하고 while 루프를 종료한다.

→ 1) end 인덱스가 totalCnt 이상인 경우

→ 2) rows.length가 pageSize보다 작은 경우 (마지막 페이지는 항상 꽉 차지 않음)

```
  start = end + 1;
```

→ 마지막이 아니라면 start 인덱스를 다음 블록의 시작 위치로 갱신하고,

→ while 루프를 계속 돌면서 다음 페이지를 요청한다.

```
} // while 끝

console.log(
  `🔚 [${ctpv}] 수집 완료 — 시도별 upsert=${localTotal}, totalCnt(추정)=${firstTotalCnt}`
);
```

→ 이 시도(ctpv)에 대한 모든 페이지 수집과 upsert가 끝난 후,

→ 해당 시도에서 총 몇 건이 upsert 되었는지, 처음에 확인한 totalCnt가 얼마였는지 로그로 정리해서 출력한다.

```
} // for CTPV_LIST 끝
console.log(
    `\n🏁 전국 registrations 완료. total upserted/modified=${totalGlobal}`
);
```

→ CTPV_LIST 전체(전국 17개 시도)에 대한 수집이 완료되면,

→ 전국 단위 총 upsert/수정 건수를 출력한다.

```
await client.close();
```

→ MongoDB 연결을 닫고 프로그램을 마무리한다.

```
main().catch((e) => {
    console.error("❌ registrations 전국 수집 실패:", e);
    process.exit(1);
});
```

→ main() 실행 중 에러가 발생하면 catch 블록에서 로그를 찍고,

→ 프로세스를 비정상 종료 코드(1)로 끝낸다.

upsert_shelters_korea.js

> upsert_shelters_korea.js는 보호소 정보 API를 통해 전국 보호센터의 기본정보(보호소명, 주소, 연락처, 위도·경도 등)를 수집하는 스크립트이다. 보호소 고유번호가 있는 경우 이를 uid로 사용하고, 없는 경우 보호소명+주소 조합으로 고유값을 생성하여 MongoDB shelters 컬렉션에 upsert한다. 또한 지도 시각화 및 유기동물–보호소 조인을 위해 위치 정보와 지역 정보가 정규화된 형태로 저장되도록 구성되어 있다.

```
import dotenv from "dotenv";
dotenv.config();
```

→ .env 파일의 환경변수를 로딩하여 process.env로 접근 가능하게 만든다.

```
import { MongoClient } from "mongodb";
```

→ MongoDB 서버 연결 및 데이터 저장을 위해 MongoClient를 불러온다.

```
const { MONGODB_URI, NAAS_KEY } = process.env;
```

→ 환경변수에서 DB URI와 공공데이터 API 키를 가져온다.

```
const BASE = "https://apis.data.go.kr/1543061/animalSheltersInfoService";
```

→ 보호소 정보 API의 기본 URL.

```
async function call(endpoint, params = {}, maxRetries = 5) {
    ...
}
```

→ 공공데이터포털 API 호출을 담당하는 함수.

→ 주요 기능:

- URL 구성(BASE + endpoint + 쿼리파라미터)
- HTTP 응답 검증(res.ok)
- 응답 JSON 파싱
- 예외 발생 시 5xx는 재시도, 4xx는 즉시 실패

  → 보호소 API는 종종 5xx 오류를 내기 때문에 재시도 로직이 필수이다.

```
async function fetchShelterInfo(pageNo, numOfRows = 1000) {
    const items = await call("shelterInfo_v2", {
        pageNo,
```

```
        numOfRows,
        _type: "json",
    });
    return items || [];
}
```

→ shelterInfo_v2 API에 pageNo, numOfRows를 붙여 요청한다.

→ 한 페이지당 1000건을 요청해 전체 페이지 수를 효율적으로 순회하도록 설계됨.

```
function normalizeShelter(r) {
    const uid = r.careRegNo ?? r.careNm + "_" + r.careAddr || null;
    return {
        uid,
        careRegNo: r.careRegNo ?? null,
        careNm: r.careNm ?? null,
        orgNm: r.orgNm ?? null,
        careAddr: r.careAddr ?? null,
        careTel: r.careTel ?? null,
        lat: r.lat ? Number(r.lat) : null,
        lng: r.lng ? Number(r.lng) : null,
        raw: r
    };
}
```

→ API에서 받은 한 레코드(r)를 내부 표준 구조로 바꾼다.

→ 핵심 포인트:

- 보호소 고유키(uid)는 careRegNo가 있으면 그걸 사용
- careRegNo가 없다면 “보호소명 + 주소” 조합으로 고유값 생성
- 위도/경도는 Number()로 변환해 정수/실수 형태로 통일
- raw에는 원본 전체 객체 저장

```
async function main() {
    const client = new MongoClient(MONGODB_URI);
    await client.connect();
    const db = client.db("animals");
    const col = db.collection("shelters");
```

→ MongoDB에 연결하고 animals DB의 shelters 컬렉션 핸들을 가져온다.

```
const first = await fetchShelterInfo(1, 1000);
const totalCnt = first.totalCnt;
```

```
console.log(`📦 전체 보호소 수: ${totalCnt}`);
```

→ 첫 페이지를 불러 totalCnt를 확인한다.

→ totalCnt는 전체 보호소 개수이며, 페이지 수 계산에 사용된다.

```
let pageNo = 1;

while (true) {
  const rows = await fetchShelterInfo(pageNo, 1000);

  if (!rows.length) {
    console.log(`➡️ pageNo=${pageNo} 더 이상 데이터 없음 → 중단`);
    break;
  }
```

→ pageNo를 1부터 증가시키며 최대 1000개씩 보호소 데이터 가져옴.

→ rows.length가 0이면 “마지막 페이지”로 판단하고 종료.

```
const docs = rows.map(normalizeShelter);

const bulkOps = docs.map((doc) => ({
  updateOne: {
    filter: { uid: doc.uid },
    update: { $set: doc },
    upsert: true,
  },
}));
```

→ 각 레코드를 normalizeShelter()로 구조 통일

→ uid 기준으로 upsert 설정

→ 같은 보호소는 항상 한 건만 유지한다.

```
await col.bulkWrite(bulkOps, { ordered: false });
```

→ 대량 upsert를 한 번에 처리하여 성능 향상

→ ordered:false라 일부 오류가 있어도 나머지 연산은 계속된다.

```
pageNo += 1;
```

→ 다음 페이지로 이동.

```
console.log(`🏁 전국 보호소 수집 완료`);
await client.close();
```

→ 수집 종료 로그 출력 후 DB 연결을 종료한다.

3. 데이터 가공 / 정제

- Pandas를 활용해 날짜형식 변환, 종·품종 분리, 성별·중성화 매핑, 지역 정보(sido/sigungu) 정규화 수행
- 몸무게, 나이, 출생연도 등 수치형 변수 파싱
- 계절(season), 요일(weekday), 월(month) 등 파생 변수 생성
- 보호소 정보와 유기동물 데이터를 보호소명 기준으로 조인하여 공간 데이터 확장

clean_animals.py

> clean_animals.py는 MongoDB에 저장된 유기동물(abandonments), 동물등록(registrations), 보호소(shelters) 데이터를 불러와 분석에 적합한 형태로 정제하는 전처리 스크립트이다. 날짜 변환, 종·품종 분리, 성별/중성화 여부 매핑, 몸무게 정규화, 출생연도·나이 추출, 지역 정보(sido, sigungu) 정렬, 계절/요일/월 파생변수 생성 등 다양한 전처리를 수행한다. 또한 유기동물 데이터와 보호소 데이터를 보호소명(careNm) 기준으로 조인한 후, 최종적으로 분석용 CSV 파일을 생성한다.

```
# clean_animals.py
import pandas as pd
import numpy as np
import os
from pymongo import MongoClient
from datetime import datetime
```

→ pandas: MongoDB에서 읽은 데이터를 DataFrame으로 다루기 위한 핵심 라이브러리

→ numpy: 결측치, 수치형 처리 등에서 사용하는 수학 라이브러리

→ os: 폴더 생성(data 디렉토리) 등 파일 시스템 작업용

→ MongoClient: MongoDB에서 원본 데이터를 읽어오기 위해 사용

→ datetime: 문자열 날짜를 datetime 객체로 바꾸거나, 연·월·요일 등 파생변수를 만들 때 사용

```
MONGO_URL = "mongodb://localhost:27017"
DB_NAME = "animals"

AB_COLLECTION = "abandonments"
REG_COLLECTION = "registrations"
SH_COLLECTION = "shelters"
```

→ MongoDB 접속 정보와 사용할 DB/컬렉션 이름을 상수로 분리해 둔 부분

→ animals DB 안에서

- AB_COLLECTION: 유기동물(abandonments)
- REG_COLLECTION: 등록현황(registrations)
- SH_COLLECTION: 보호소(shelters)

  를 읽어와서 전처리하게 된다.

```
def main():
    # 1) MongoDB 연결 및 컬렉션 로드
    client = MongoClient(MONGO_URL)
    db = client[DB_NAME]

    ab_raw = pd.DataFrame(list(db[AB_COLLECTION].find({})))
    reg_raw = pd.DataFrame(list(db[REG_COLLECTION].find({})))
    sh_raw = pd.DataFrame(list(db[SH_COLLECTION].find({})))

    # 2) 각 컬렉션별 전처리
    ab_clean = clean_abandonments(ab_raw)
    reg_clean = clean_registrations(reg_raw)
    sh_clean = clean_shelters(sh_raw)

    # 3) 유기동물 + 보호소 정보 조인
    ab_merged = merge_abandonments_with_shelters(ab_clean, sh_clean)

    # 4) CSV로 저장
    os.makedirs("data", exist_ok=True)
    print("CSV로 저장 중...")
    ab_merged.to_csv("data/clean_abandonments.csv", index=False)
    reg_clean.to_csv("data/clean_registrations.csv", index=False)
    sh_clean.to_csv("data/clean_shelters.csv", index=False)

    print("저장 완료!")
    print(" - clean_abandonments.csv")
    print(" - clean_registrations.csv")
    print(" - clean_shelters.csv")
```

→ 전체 흐름 핵심 요약

1. MongoDB에서 세 컬렉션을 DataFrame으로 불러온다.
2. 유기동물/등록현황/보호소 각각에 대해 **전처리 함수**를 따로 돌린다.
3. 유기동물 + 보호소를 **보호소 이름(careNm)** 기준으로 조인해서 위치·주소를 붙인다.
4. 최종 결과를 `data/` 폴더 아래 CSV 3개로 저장한다.

```
def clean_abandonments(df):
    # 1) 날짜 컬럼 변환
    date_cols = ["happenDt", "noticeSdt", "noticeEdt"]
    for col in date_cols:
        df[col] = pd.to_datetime(df[col], errors="coerce", format="%Y-%m-%d")

    # 2) kindCd에서 종(species) / 품종(kindName) 분리
    def split_kind(kind):
        if pd.isna(kind):
            return pd.Series([np.nan, np.nan])
        m = re.match(r"\[(.+)\]\s*(.*)", str(kind))
        if m:
            return pd.Series([m.group(1), m.group(2)])
        else:
            return pd.Series([np.nan, kind])

    df[["species", "kindName"]] = df["kindCd"].apply(split_kind)

    # 3) 성별 코드 -> 사람이 읽기 쉬운 값
    sex_map = {"M": "Male", "F": "Female"}
    df["sex"] = df["sexCd"].map(sex_map).fillna("Unknown")

    # 4) 중성화 여부 코드 -> 범주형 텍스트
    neuter_map = {"Y": "Neutered", "N": "Not Neutered"}
    df["neuter"] = df["neuterYn"].map(neuter_map).fillna("Unknown")

    # 5) 몸무게 "6.8(Kg)" -> 숫자 6.8
    df["weight"] = (
        df["weight"]
        .astype(str)
        .str.extract(r"([\d\.]+)", expand=False)
        .astype(float)
    )

    # 6) 출생연도 / 나이 계산 (age)
    df["birthYear"] = (
        df["age"].astype(str).str.extract(r"(\d{4})", expand=False).astype(float)
    )
    current_year = datetime.now().year
    df["age_years"] = current_year - df["birthYear"]

    # 7) 시도/시군구 분리 (주소 or orgNm 활용)
    df["sido"] = df["orgNm"].str.split().str[0]
    df["sigungu"] = df["orgNm"].str.split().str[1]

    # 8) 날짜에서 월/요일/계절 파생변수 만들기
    df["month"] = df["happenDt"].dt.month
```

```
    df["weekday"] = df["happenDt"].dt.weekday  # 0=월, 6=일

    def to_season(m):
        if m in [3, 4, 5]:
            return "Spring"
        elif m in [6, 7, 8]:
            return "Summer"
        elif m in [9, 10, 11]:
            return "Fall"
        else:
            return "Winter"

    df["season"] = df["month"].apply(to_season)

    return df
```

→ 전처리 포인트 설명

1. **날짜 문자열 → datetime**
   - `happenDt`, `noticeSdt`, `noticeEdt` 를 `to_datetime`으로 변환
   - `errors="coerce"`로 이상한 값은 NaT로 처리해서 이후 분석에서 제외 가능
2. **kindCd → species / kindName 분리**
   - 예: `"[개] 믹스견"` → species="개", kindName="믹스견"
   - 나중에 "종별/품종별 유기현황" 분석에 쓰기 쉽게 만든다.
3. **성별/중성화 코드 정리**
   - sexCd: M, F, Q ... → "Male", "Female", "Unknown"
   - neuterYn: Y, N, U → "Neutered", "Not Neutered", "Unknown"
   - 그래프나 보고서에 바로 쓰기 좋은 형태로 변환
4. **몸무게 정규화**
   - `"6.8(Kg)"`, `"6 Kg"` 등에서 숫자 부분만 정규식으로 추출
   - float 타입으로 바꿔 통계/모델링에 사용 가능
5. **출생연도/나이 계산**
   - age 문자열에서 4자리 연도만 뽑아 `birthYear`로 저장
   - `현재연도 - birthYear` → 나이 추정값(`age_years`) 생성
6. **지역 정보 분리**
   - orgNm(예: "인천광역시 부평구") 또는 주소에서
   - 첫 단어 → sido, 두 번째 단어 → sigungu 로 쪼개서 지역 분석에 활용
7. **시계열 파생변수**
   - month, weekday, season 컬럼을 만들어
   - 월별·계절별 패턴, 요일별 패턴 분석에 바로 쓰도록 한다.

```
def clean_registrations(df):
    # 1) 시도/시군구 이름
    df["sido"] = df["sido"].fillna(df["raw"].apply(lambda r: r.get("CTPV")))
    df["sigungu"] = df["sigungu"].fillna(df["raw"].apply(lambda r: r.get("SGG")))

    # 2) 출생연도
    df["birthYear"] = df["birthYear"].fillna(
        df["raw"].apply(lambda r: str(r.get("BRDT", ""))[:4])
    ).astype(float)

    # 3) RFID/종/품종
    df["rfidType"] = df["rfidType"].fillna(
        df["raw"].apply(lambda r: r.get("RFID_SE"))
    )
    df["species"] = df["species"].fillna(
        df["raw"].apply(lambda r: r.get("LVSTCK_KND"))
    )
    df["kind"] = df["kind"].fillna(
        df["raw"].apply(lambda r: r.get("SPCS"))
    )

    # 4) count 숫자형 변환
    df["count"] = (
        df["count"]
        .fillna(df["raw"].apply(lambda r: r.get("CNT", 0)))
        .astype(str)
        .str.replace(",", "", regex=False)
        .astype(float)
    )

    return df
```

→ 포인트 설명

1. **시도/시군구**
   - Mongo에 저장된 sido/sigungu가 비어 있으면 원본 raw(CTPV, SGG)에서 다시 가져와 채운다.
   - 이 덕분에 등록현황도 유기데이터와 동일한 지역 필드 기준으로 비교 가능.
2. **출생연도**
   - BRDT가 `YYYYMMDD` 또는 연도만 들어올 수 있어 앞 4자리만 잘라 사용.
   - float으로 변환해 연속형 변수처럼 다룰 수 있다.
3. **RFID, 축종, 품종**
   - raw의 RFID_SE, LVSTCK_KND, SPCS에서 가져와 rfidType/species/kind에 통일
   - 이후 groupby 시 컬럼명이 일관되도록 정리하는 작업.

4. **count 정리**
    - CNT에 쉼표가 포함되어 있을 수 있으므로 문자열로 바꿔 쉼표 제거 후 float으로 변환
    - 이렇게 해야 합계(sum)·비율 계산이 정확하게 동작한다.
    -

```
def clean_shelters(df):
    # 1) 주소에서 시도/시군구 추출
    df["sido"] = df["careAddr"].str.split().str[0]
    df["sigungu"] = df["careAddr"].str.split().str[1]

    # 2) 위도/경도 통합
    df["lat"] = df["lat"].fillna(
        df["raw"].apply(lambda r: float(r.get("lat") or r.get("LAT") or np.nan))
    )
    df["lng"] = df["lng"].fillna(
        df["raw"].apply(lambda r: float(r.get("lng") or r.get("LNG") or np.nan))
    )

    # 3) 지정일자(개설일) 변환
    df["openDate"] = pd.to_datetime(
        df["openDate"].fillna(df["raw"].apply(lambda r: r.get("dsignationDate"))),
        errors="coerce",
    )

    return df
```

→ 포인트 설명

1. **주소에서 시도/시군구 추출**
    - careAddr 예: `"인천광역시 부평구 ***로 123"`
    - 첫 단어: "인천광역시" → sido
    - 두 번째 단어: "부평구" → sigungu
    - 이렇게 하면 보호소도 유기데이터와 동일한 지역 기준으로 묶을 수 있다.
2. **위도/경도 통합**
    - API/시스템에 따라 lat/LAT/latitude 등 필드 이름이 제각각일 수 있어서,
    - raw 안의 여러 필드 중 하나라도 있으면 가져와 float으로 변환
    - 지도 시각화이나 거리 계산에 쓰기 위해 숫자형으로 통일.
3. **개설일(openDate)**
    - openDate 컬럼이 비어 있으면 raw.dsignationDate 같은 다른 필드에서 가져옴
    - to_datetime으로 변환해 보호소 개설 시기 분석 등에 활용할 수 있게 한다.

```
def merge_abandonments_with_shelters(ab_df, sh_df):
    # 보호소 이름(careNm)을 기준으로 left join
    merged = ab_df.merge(
        sh_df[["careNm", "sido", "sigungu", "lat", "lng"]],
        on="careNm",
        how="left",
        suffixes=("", "_shelter"),
    )
    return merged
```

→ 포인트 설명

- 유기동물 데이터(ab_df)에 보호소 정보(sh_df)를 **careNm(보호소 이름)** 으로 조인한다.
- 보호소의 sido/sigungu/좌표(lat, lng)를 유기동물 데이터에 붙여서,

  “어떤 보호소에 얼마나 많은 유기동물이 들어와 있는지”,

  “보호소 위치 기준 유기 패턴” 등을 분석할 수 있게 만든다.

```
os.makedirs("data", exist_ok=True)
print("CSV로 저장 중...")
ab_merged.to_csv("data/clean_abandonments.csv", index=False)
reg_clean.to_csv("data/clean_registrations.csv", index=False)
sh_clean.to_csv("data/clean_shelters.csv", index=False)

print("저장 완료!")
print(" - clean_abandonments.csv")
print(" - clean_registrations.csv")
print(" - clean_shelters.csv")
```

→ data 폴더가 없으면 생성하고,

→ 전처리/조인된 결과를 CSV 세 개로 저장한다.

→ 이후 모든 시각화(visualization.py)와 통계/모델링(analysis_model.py)은 이 CSV를 기반으로 진행된다.

```
if __name__ == "__main__":
    main()
```

→ 이 파일을 **직접 실행했을 때만** main()을 호출해서 전체 전처리 파이프라인을 수행한다.

→ 다른 모듈에서 import할 때는 자동으로 실행되지 않는다.

4. 데이터 분석

- 연도·월·계절별 유기동물 발생 추이 분석(Time Series)
- 시도·시군구별 유기·등록 패턴 비교(Spatial Analysis)

- 성별·중성화 여부·종/품종·나이 등 다변수 상관 분석 수행
- RandomForest 기반 '처리결과(processState)' 예측 모델 구축 및 성능 평가

analysis_model.py

> analysis_model.py는 전처리된 유기동물 데이터에 대해 기본 통계 분석과 머신러닝 기반 처리결과 예측 모델(RandomForest)을 수행하는 스크립트이다. 종·성별·중성화 여부·나이·몸무게·계절·요일·지역 등 다양한 변수를 입력_features로 사용하여 processState(입양/반환/안락사 등)를 예측한다. 또한 연도/월별 유기동물 추이, 지역별 유기 건수, 성별/중성화 여부/종에 따른 처리결과 교차표 등 여러 형태의 분석 지표를 출력하여 데이터 특성을 파악하는 역할을 한다.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, classification_report
from datetime import datetime
import os
```

→ 데이터 분석 및 모델링에 필요한 라이브러리들을 불러온다.

→ RandomForest, OneHotEncoder, train/test split 등 핵심 ML 도구 포함.

```
DATA_PATH = "data/clean_abandonments.csv"
```

→ 모델 학습에 사용할 데이터는 **전처리된 유기동물 CSV**.

```
class Tee:
    def __init__(self, filename):
        self.file = open(filename, "w", encoding="utf-8")
        self.stdout = sys.stdout

    def write(self, data):
        self.file.write(data)
        self.stdout.write(data)

    def flush(self):
        self.file.flush()
```

→ print 결과를 **콘솔 + 파일** 두 곳에 동시에 출력하기 위한 유틸.

→ 모델 학습 로그를 그대로 파일로 남기기 때문에 보고서에 수치를 인용할 수 있다.

```
def analyze_basic_patterns(ab):
    print("▶ 종 분포:", ab["species"].value_counts())
    print("▶ 성별 분포:", ab["sex"].value_counts())
    print("▶ 중성화 분포:", ab["neuter"].value_counts())
    print("▶ 나이 통계:")
    print(ab["age_years"].describe())
```

→ EDA의 핵심 통계만 빠르게 출력한다.

→ 종 분포 / 성별 / 중성화 비율 / 나이 통계 등

전처리된 데이터가 잘 들어왔는지 확인하는 단계.

```
def build_and_evaluate_model(ab):
    df = ab.sample(frac=0.1, random_state=42)
```

→ 데이터가 수십만 건이기 때문에 **10% 샘플링**하여 모델 학습 시간 단축.

```
features = ["species", "kindName", "sex", "neuter",
            "season", "weekday", "sido", "sigungu",
            "age_years", "weight", "month"]

target = "processState"
```

→ 입력 특징(개체 특성 + 계절/요일 + 지역)과

→ 예측하려는 목표값(처리결과: 입양·반환·안락사 등)을 정의한다.

```
categorical = ["species", "kindName", "sex", "neuter",
               "season", "weekday", "sido", "sigungu"]
numeric = ["age_years", "weight", "month"]
```

→ 범주형/수치형 컬럼을 분리해 OneHotEncoder + 그대로 전달로 처리하기 위함.

```
preprocess = ColumnTransformer(
    [
        ("cat", OneHotEncoder(handle_unknown="ignore"), categorical),
        ("num", "passthrough", numeric),
    ]
)
```

→ 범주형은 원핫인코딩, 수치형은 그대로 사용하도록 구성한 전처리기.

```
model = RandomForestClassifier(
    n_estimators=200, random_state=42, n_jobs=-1
)
```

→ 다중 클래스 분류(RandomForest).

→ n_jobs=-1로 모든 CPU 코어를 사용하여 학습 속도 최적화.

```
pipe = Pipeline(
    steps=[("preprocess", preprocess), ("model", model)]
)
```

→ 전처리 + 모델 학습 과정을 하나의 파이프라인으로 통합.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

→ 80:20으로 데이터를 나누고, stratify=y로 각 클래스 비율을 유지.

→ 특히 처리결과(processState) 클래스가 불균형이라 stratify가 매우 중요.

```
pipe.fit(X_train, y_train)
```

→ 파이프라인 전체 학습 수행.

```
pred = pipe.predict(X_test)
print("정확도:", accuracy_score(y_test, pred))
print(classification_report(y_test, pred))
```

→ 모델의 예측 정확도(accuracy)와

→ 각 클래스별 정밀도/재현율/F1 점수 출력.

→ 보고서에서는 이 부분의 수치를 그대로 인용해서

“입양 클래스는 OO% 성능, 반려/기증 클래스는 낮음” 같은 분석 가능.

```
def analyze_time_series(ab):
    print(ab.groupby("happenYear")["uid"].count())
    print(ab.groupby("happenMonth")["uid"].count())
    print(pd.crosstab(ab["happenYear"], ab["processState"]))
```

→ 연도별 유기 건수

→ 월(month)별 유기 건수

→ 연도 × 처리결과 교차표

→ 이런 수치들은 시각화 없이도 트렌드를 확인할 수 있음

(예: 여름철 증가, 코로나 기간 변화 등)

```
def analyze_spatial(ab):
    print(ab.groupby("sido")["uid"].count().sort_values(ascending=False).head(10))
    print(ab.groupby("sigungu")["uid"].count().sort_values(ascending=False).head(20))
```

→ 시도별 TOP 10 / 시군구별 TOP 20 유기 건수를 뽑는다.

→ 지역별 유기동물 집중 지역을 파악하는 데 사용.

```
print(pd.crosstab(ab["sido"], ab["processState"]))
```

→ 지역별 처리결과 차이를 확인.

```
def analyze_correlations(ab):
    print(ab[["age_years", "weight", "month"]].corr())
    print(pd.crosstab(ab["sex"], ab["processState"]))
    print(pd.crosstab(ab["neuter"], ab["processState"]))
    print(pd.crosstab(ab["species"], ab["season"]))
```

→ 수치형(age, weight, month)의 Pearson 상관계수 출력

→ 성별 × 처리결과

→ 중성화 × 처리결과

→ 종 × 계절

→ 이런 교차표는 정책적 인사이트 도출에 활용 가능

```
ab = pd.read_csv(DATA_PATH)
analyze_basic_patterns(ab)
build_and_evaluate_model(ab)
analyze_time_series(ab)
analyze_spatial(ab)
analyze_correlations(ab)
```

→ 파일 한 번 실행하면

1. 기본 EDA
2. RandomForest 모델 학습/평가

3. 시계열 분석
4. 공간(지역) 분석
5. 상관분석

   까지 한 번에 수행하는 구조.

분석 결과 (clean_abandonments.csv log)

📥 데이터 로드 중: data/clean_abandonments.csv

- 유기동물 데이터 shape: (1940162, 44)

===== [4-1] 처리결과 분포 (processState 비율) =====
processState
종료(안락사) 0.2924
종료(자연사) 0.2696
종료(입양) 0.2506
종료(반환) 0.1496
종료(기증) 0.0272
종료(방사) 0.0060
보호중 0.0044
종료(기타) 0.0001
Name: proportion, dtype: float64

===== [4-1] 종(species) 분포 =====
species
개 1351991
고양이 532343
기타축종 55828
Name: count, dtype: int64

===== [4-1] 성별(sex) 분포 =====
sex
Male 950482
Female 891316
Unknown 98364
Name: count, dtype: int64

===== [4-1] 중성화(neuter) 여부 분포 =====
neuter
0.0 1041691
NaN 770965
1.0 127506
Name: count, dtype: int64

===== [4-1] 연령(age) 기초 통계 =====
count 1.940005e+06
mean 4.810638e+00
std 2.945632e+00
min 0.000000e+00
25% 3.000000e+00

50% 4.000000e+00
75% 6.000000e+00
max 3.300000e+01
Name: age, dtype: float64

===== [4-2] 예측 모델 구축 (processState 예측) =====

🔍 10% 샘플링 적용 중... (모델 학습 전용)

- 샘플링 후 데이터 shape: (194016, 44)
- 학습 데이터 크기: (155212, 11)
- 테스트 데이터 크기: (38804, 11)

[모델 정확도] accuracy: 0.8092

[클래스별 성능 지표] classification_report:
precision recall f1-score support

```
     보호중       0.68      0.27      0.39       171
   종료(기증)       0.89      0.62      0.73      1040
   종료(기타)       1.00      1.00      1.00         5
   종료(반환)       0.82      0.82      0.82      5794
   종료(방사)       0.60      0.25      0.35       231
  종료(안락사)       0.84      0.88      0.86     11321
   종료(입양)       0.76      0.74      0.75      9769
  종료(자연사)       0.81      0.83      0.82     10473

  accuracy                           0.81     38804
```

macro avg 0.80 0.68 0.72 38804
weighted avg 0.81 0.81 0.81 38804

===== [4-3] 시계열 분석 (Time Series) =====

[연도별 유기동물 발생 건수]
year
2020 400431
2021 378361
2022 371741
2023 406595
2024 383034
Name: uid, dtype: int64

[연도 × 처리결과 발생 건수]
year processState
2020 보호중 240
종료(기증) 8405
종료(기타) 66
종료(반환) 62485
종료(방사) 1921
종료(안락사) 122693
종료(입양) 109489

```
종료(자연사)         95132
2021  보호중             691
종료(기증)           8397
종료(기타)             65
종료(반환)          59218
종료(방사)           1986
종료(안락사)         98427
종료(입양)         113683
종료(자연사)         95894
2022  보호중            1052
종료(기증)          11342
종료(기타)             87
종료(반환)          56853
Name: uid, dtype: int64

[월별 유기동물 발생 건수]
month
1     132681
2     119361
3     139005
4     154419
5     196899
6     192279
7     202480
8     179877
9     167827
10    179688
11    147397
12    128249
Name: uid, dtype: int64

[월별 × 처리결과 발생 건수]
month  processState
1      보호중             368
종료(기증)          5903
종료(기타)             2
종료(반환)         19314
종료(방사)           472
종료(안락사)        48158
종료(입양)         34640
종료(자연사)        23824
2      보호중             364
종료(기증)          3041
종료(기타)             46
종료(반환)         19363
종료(방사)           605
종료(안락사)        42799
종료(입양)         32417
종료(자연사)        20726
```

3 보호중 408
종료(기증) 4456
종료(기타) 24
종료(반환) 22409
Name: uid, dtype: int64

===== [4-4] 지역 기반 분석 (Spatial Analysis) =====

[시도별 유기동물 발생 건수 TOP 10]
sido
경기도 815129
서울특별시 500328
광주광역시 92985
부산광역시 64644
경상남도 61879
울산광역시 54764
전라남도 47779
경상북도 45016
전북특별자치도 43517
충청남도 42319
Name: uid, dtype: int64

[시군구별 유기동물 발생 건수 TOP 20]
sigungu
포천시 173275
김포시 134415
양주시 99934
연천군 74762
파주시 71745
의정부시 52693
북구 50948
강서구 40454
은평구 38778
강북구 37149
중구 36504
서구 34263
동두천시 31059
중랑구 30455
남구 29430
성북구 29406
송파구 29232
노원구 27946
안산시 27586
동대문구 27423
Name: uid, dtype: int64

[시도 × 처리결과 교차표]
processState 보호중 종료(기증) 종료(기타) 종료(반환) 종료(방사) 종료(안락사) 종료(입양) 종료(자연사)
sido

강원특별자치도 772 179 0 4204 445 6526 8281 8475
경기도 1377 22808 37 102820 2122 370989 170354 144622
경상남도 843 2009 0 5308 733 14861 17381 20744
경상북도 626 407 4 4226 438 10987 18647 9681
광주광역시 799 2251 6 13116 288 9994 22615 43916
대구광역시 46 9 0 4523 172 5437 10988 14582
대전광역시 74 34 0 3612 35 1513 3304 1876
부산광역시 585 403 57 6104 505 1919 16174 38897
서울특별시 971 17506 61 117944 805 83015 133036 146990
세종특별자치시 4 4 0 63 85 817 925 462

===== [4-5] 다변수 상관관계 분석 =====

[수치형 변수 상관계수]
age weight month
age 1.000 -0.0 -0.017
weight -0.000 1.0 -0.000
month -0.017 -0.0 1.000

[성별(sex) × 처리결과(processState) 교차표]
processState 보호중 종료(기증) 종료(기타) 종료(반환) 종료(방사) 종료(안락사) 종료(입양) 종료(자연사)
sex

Female 4308 24531 89 121934 4553 278167 226638 231096
Male 3798 25592 54 164332 4761 279111 231036 241798
Unknown 522 2745 75 3914 2357 10062 28534 50155

[중성화(neuter) × 처리결과(processState) 교차표]
processState 보호중 종료(기증) 종료(기타) 종료(반환) 종료(방사) 종료(안락사) 종료(입양) 종료(자연사)
neuter

0.0 5123 29072 69 114283 7579 299132 264704 321729
1.0 908 4868 1 57535 980 15061 34448 13705

[종(species) × 계절(season) 교차표]
season Fall Spring Summer Winter
species

개 326135 343964 361069 320823
고양이 152611 133402 195555 50775
기타축종 16166 12957 18012 8693

analysis_log_20251125_144656.txt

5. 데이터 시각화

- Matplotlib·Seaborn 기반 시각화(선그래프, 막대그래프, 파이차트, Heatmap 등)
- folium 기반 전국 보호소 지도(HTML) 생성
- 등록 대비 유기 비율(1000마리당 유기) 비교 그래프 제작

- Tableau용 집계 데이터셋 생성(시·도×월×처리결과, 출생연도별 등록현황 등)

visualization.py

> visualization.py는 전처리된 CSV 데이터를 기반으로 다양한 시각화를 생성하는 스크립트이다. 유기동물 데이터에 대해 연도·월·계절별 시계열 분석, 시도×월 Heatmap, 처리결과 비율과 연도별 변화, 종·품종 분포 등을 그래프로 출력한다. 등록현황 데이터에서는 출생연도별 등록두수와 시도별 등록 상위 지역을 시각화하고, 등록 대비 유기 비율도 함께 비교한다. 보호소 데이터에 대해서는 지역별 보호소 수와 위경도 기반 지도 시각화를 생성한다. 모든 시각화 결과는 figures 폴더에 PNG/HTML로 저장된다.

```
# src/visualization.py
import os
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import folium

DATA_DIR = "data"
AB_PATH = os.path.join(DATA_DIR, "clean_abandonments.csv")
REG_PATH = os.path.join(DATA_DIR, "clean_registrations.csv")
SH_PATH = os.path.join(DATA_DIR, "clean_shelters.csv")
FIG_DIR = "figures"
```

→ 전처리된 CSV 경로(유기동물, 등록현황, 보호소)와

→ 그래프를 저장할 폴더(FIG_DIR)를 정의해놓은 부분.

→ 이 파일에서 그리는 모든 그림은 figures/ 폴더에 PNG/HTML로 저장된다.

```
def setup_env():
    os.makedirs(FIG_DIR, exist_ok=True)

    sns.set(style="whitegrid")
    plt.rcParams["axes.unicode_minus"] = False

    try:
        plt.rc("font", family="Malgun Gothic")
    except Exception:
        pass
```

→ figures 폴더가 없으면 생성.

→ seaborn 스타일을 whitegrid로 지정.

→ 한글(말굽고딕) 설정 + 마이너스 깨짐 방지.

→ 한글 그래프 제목/축 라벨이 깨지지 않게 하는 설정이다.

```
def load_data():
    print(f"📥 유기동물 데이터 로드: {AB_PATH}")
    ab = pd.read_csv(AB_PATH)
    print(" - abandonments shape:", ab.shape)

    print(f"📥 등록현황 데이터 로드: {REG_PATH}")
    reg = pd.read_csv(REG_PATH)
    print(" - registrations shape:", reg.shape)

    print(f"📥 보호소 데이터 로드: {SH_PATH}")
    sh = pd.read_csv(SH_PATH)
    print(" - shelters shape:", sh.shape)

    return ab, reg, sh
```

→ 전처리 단계에서 만든 **clean_abandonments / clean_registrations / clean_shelters** 3개 CSV를 읽어서 DataFrame으로 로드.

→ 각 데이터의 shape를 출력해, 전처리 결과를 확인할 수 있도록 한다.

```
def plot_time_series_ab(ab: pd.DataFrame):
    print("\n📊 [유기동물 시계열] 연도·월·계절별 그래프 생성 중...")

    # 연도별
    if "year" in ab.columns:
        yearly = ab.groupby("year")["uid"].count()
        plt.figure(figsize=(8, 4))
        yearly.plot(marker="o")
        plt.title("연도별 유기동물 발생 추이")
        plt.xlabel("연도")
        plt.ylabel("건수")
        plt.tight_layout()
        plt.savefig(os.path.join(FIG_DIR, "ab_timeseries_yearly.png"), dpi=200)
        plt.close()
```

→ `ab_timeseries_yearly.png`

→ 연도(year)별 유기동물 건수 추이를 선 그래프로 그림.

→ 유기 건수가 **어느 해에 많았는지** 한눈에 보는 용도.

```
    # 월별
    monthly = ab.groupby("month")["uid"].count().reindex(range(1, 13))
    plt.figure(figsize=(8, 4))
    monthly.plot(kind="line", marker="o")
    plt.title("월별 유기동물 발생 추이")
    plt.xlabel("월")
```

```
    plt.ylabel("건수")
    plt.xticks(range(1, 13))
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "ab_timeseries_monthly.png"), dpi=200)
    plt.close()
```

→ `ab_timeseries_monthly.png`

→ 1~12월별 유기 건수를 선그래프로 표현.

→ **여름/겨울에 유기가 증가하는지** 등 계절성 확인.

```
    # 계절별
    if "season" in ab.columns:
        season_order = ["Spring", "Summer", "Fall", "Winter"]
        season = ab.groupby("season")["uid"].count().reindex(season_order)
        plt.figure(figsize=(6, 4))
        season.plot(kind="bar")
        plt.title("계절별 유기동물 발생 건수")
        plt.xlabel("계절")
        plt.ylabel("건수")
        plt.tight_layout()
        plt.savefig(os.path.join(FIG_DIR, "ab_timeseries_season.png"), dpi=200)
        plt.close()
```

→ `ab_timeseries_season.png`

→ 봄/여름/가을/겨울 **계절별 유기 건수**를 막대그래프로 표현.

```
def plot_heatmap_sido_month_ab(ab: pd.DataFrame):
    print("\n📊 [Heatmap] 시도 × 월 유기동물 패턴 시각화 중...")

    heat = ab.groupby(["sido", "month"])["uid"].count().unstack(fill_value=0)

    plt.figure(figsize=(12, 8))
    sns.heatmap(
        heat,
        cmap="Blues",
        linewidths=0.3,
        linecolor="lightgrey",
        cbar_kws={"label": "유기동물 건수"},
    )
    plt.title("시도 × 월별 유기동물 발생 Heatmap")
    plt.xlabel("월")
    plt.ylabel("시도")
    plt.tight_layout()
```

```
    plt.savefig(os.path.join(FIG_DIR, "ab_heatmap_sido_month.png"), dpi=200)
    plt.close()
```

→ `ab_heatmap_sido_month.png`

→ “시도 × month” 행렬 위에 유기 건수를 색상으로 표현.

→ 어느 시도에서 **어느 달에 유기가 집중되는지** 직관적으로 보이는 그림.

```
def plot_registrations(reg: pd.DataFrame):
    print("\n📊 [등록현황] 출생연도·지역별 등록 패턴 시각화 중...")

    # 출생연도 기준 등록두수 (birthYear가 있다고 가정)
    if "birthYear" in reg.columns:
        reg_clean = reg.copy()
        reg_clean["birthYear"] = pd.to_numeric(reg_clean["birthYear"], errors="coerce")
        reg_clean = reg_clean[reg_clean["birthYear"].between(1900, 2030)]

        yearly = (
            reg_clean.groupby("birthYear")["count"]
            .sum()
            .sort_index()
        )

        plt.figure(figsize=(8, 4))
        yearly.plot(marker="o")
        plt.title("출생연도별 등록두수 추이 (1900~2030)")
        plt.xlabel("출생연도")
        plt.ylabel("등록 마릿수")
        plt.xlim(1990, 2030)
        plt.tight_layout()
        plt.savefig(os.path.join(FIG_DIR, "reg_by_birthyear.png"), dpi=200)
        plt.close()
```

→ `reg_by_birthyear.png`

→ 출생연도별 등록 마릿수 추이를 그린 선그래프.

→ “2010년대 이후 출생 개체 등록이 급증하는지?” 같은 패턴 확인.

```
    # 시도별 등록두수 TOP 10
    reg_sido = reg.groupby("sido")["count"].sum().sort_values(ascending=False)
    plt.figure(figsize=(8, 4))
    reg_sido.head(10).plot(kind="bar")
    plt.title("시도별 등록 마릿수 TOP 10")
    plt.xlabel("시도")
    plt.ylabel("등록 마릿수")
    plt.xticks(rotation=45, ha="right")
```

```
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "reg_sido_top10.png"), dpi=200)
    plt.close()
```

→ `reg_sido_top10.png`

→ 등록 마릿수가 많은 **상위 10개 시도** 막대그래프.

```
def plot_abandon_vs_register_by_sido(ab: pd.DataFrame, reg: pd.DataFrame):
    print("\n📊 [등록 vs 유기] 시도 단위 비교 그래프 생성 중...")

    ab_sido = ab.groupby("sido")["uid"].count().rename("abandon_cnt")
    reg_sido = reg.groupby("sido")["count"].sum().rename("reg_cnt")

    merged = pd.concat([ab_sido, reg_sido], axis=1).dropna()
    merged = merged[merged["reg_cnt"] > 0]

    merged["abandon_per_1000"] = merged["abandon_cnt"] / merged["reg_cnt"] * 1000
    top = merged.sort_values("abandon_per_1000", ascending=False).head(10)
```

→ 시도별로

- 유기 건수(abandon_cnt)
- 등록두수(reg_cnt)
- 1,000마리당 유기 비율(abandon_per_1000)

  을 계산해서, 비율 기준 상위 10개 시도를 뽑는다.

```
    # ① 등록 vs 유기 건수 (이중 바차트)
    plt.figure(figsize=(10, 5))
    idx = range(len(top))
    width = 0.35

    plt.bar([i - width/2 for i in idx], top["reg_cnt"], width=width, label="등록두수")
    plt.bar([i + width/2 for i in idx], top["abandon_cnt"],
    width=width, label="유기건수")

    plt.xticks(idx, top.index, rotation=45, ha="right")
    plt.title("시도별 등록두수 vs 유기건수 (TOP 10)")
    plt.ylabel("건수")
    plt.legend()
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "sido_reg_vs_abandon.png"), dpi=200)
    plt.close()
```

→ `sido_reg_vs_abandon.png`

→ 시도별 등록·유기 건수를 **나란한 막대그래프**로 비교.

```
    # ② 1,000마리당 유기 비율
    plt.figure(figsize=(10, 5))
    top["abandon_per_1000"].plot(kind="bar")
    plt.title("시도별 1,000마리당 유기 발생 비율 (등록 대비)")
    plt.xlabel("시도")
    plt.ylabel("유기 건수 / 1,000마리")
    plt.xticks(rotation=45, ha="right")
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "sido_abandon_per_1000.png"), dpi=200)
    plt.close()
```

→ `sido_abandon_per_1000.png`

→ "등록 1,000마리당 유기 몇 건?" 비율 기준으로 시도를 비교.

→ 등록 대비 유기 비율이 높은 문제 지역을 찾는 용도.

```
def plot_shelter_distribution(sh: pd.DataFrame):
    print("\n🗺️ [보호소 분포] 보호소 위치·지역별 개수 시각화 중...")

    # 시도/지자체별 보호소 개수 (막대그래프)
    if "orgNm" in sh.columns:
        sh_sido = sh.groupby("orgNm")["uid"].count().sort_values(ascending=False)
        plt.figure(figsize=(8, 4))
        sh_sido.head(15).plot(kind="bar")
        plt.title("지자체(orgNm)별 보호소 수 TOP 15")
        plt.xlabel("지자체")
        plt.ylabel("보호소 수")
        plt.xticks(rotation=60, ha="right")
        plt.tight_layout()
        plt.savefig(os.path.join(FIG_DIR, "sh_orgNm_top15.png"), dpi=200)
        plt.close()
```

→ `sh_orgNm_top15.png`

→ 지자체(orgNm) 기준 보호소 개수 상위 15개 지역.

```
    # ① Matplotlib 위경도 스캐터
    if {"lat", "lng"}.issubset(sh.columns):
        geo = sh.dropna(subset=["lat", "lng"])
        plt.figure(figsize=(8, 8))
        plt.scatter(geo["lng"], geo["lat"], s=20, alpha=0.6)
        plt.title("전국 보호소 위치 분포 (위경도 스캐터)")
        plt.xlabel("경도(lng)")
```

```
    plt.ylabel("위도(lat)")
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "sh_map_shelters.png"), dpi=200)
    plt.close()
```

→ sh_map_shelters.png

→ 위도/경도를 이용해 전국 보호소 위치를 점으로 찍은 평면 scatter.

```
    # ② folium 지도 위에 찍기 (인터랙티브 지도)
    print("   - folium 기반 HTML 지도 생성 중...")

    center_lat = geo["lat"].mean()
    center_lng = geo["lng"].mean()

    m = folium.Map(
        location=[center_lat, center_lng],
        zoom_start=7,
        tiles="CartoDB positron",
    )

    for _, row in geo.iterrows():
        folium.CircleMarker(
            location=[row["lat"], row["lng"]],
            radius=3,
            fill=True,
            fill_opacity=0.7,
            popup=row.get("careNm", ""),
        ).add_to(m)

    html_path = os.path.join(FIG_DIR, "sh_map_shelters.html")
    m.save(html_path)
    print(f"   - 보호소 지도 HTML 저장 완료: {html_path}")
```

→ sh_map_shelters.html

→ folium 기반 인터랙티브 지도.

→ 마우스로 확대/이동 가능, 마커 클릭 시 보호소 이름 팝업.

```
def plot_process_state(ab: pd.DataFrame):
    print("\n📊 [처리결과] 상태별 비율 / 연도별 비율 추이 시각화 중...")

    state_ratio = ab["processState"].value_counts(normalize=True)
    plt.figure(figsize=(6, 6))
    state_ratio.plot(kind="pie", autopct="%.1f%%")
    plt.ylabel("")
```

```
    plt.title("처리결과(processState) 비율")
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "ab_process_state_ratio.png"), dpi=200)
    plt.close()
```

→ `ab_process_state_ratio.png`

→ 입양/반환/안락사/기타 등 처리결과 비율을 파이차트로 표현.

```
    if "year" in ab.columns:
        year_state = (
            ab.groupby(["year", "processState"])["uid"]
            .count()
            .unstack(fill_value=0)
        )
        year_state_ratio = year_state.div(year_state.sum(axis=1), axis=0)

        plt.figure(figsize=(10, 5))
        plt.stackplot(
            year_state_ratio.index,
            year_state_ratio.T.values,
            labels=year_state_ratio.columns,
        )
        plt.legend(loc="upper left", bbox_to_anchor=(1.02, 1.0))
        plt.title("연도별 처리결과 비율 추이 (stacked area)")
        plt.xlabel("연도")
        plt.ylabel("비율")
        plt.tight_layout()
        plt.savefig(
            os.path.join(FIG_DIR, "ab_year_process_state_ratio.png"), dpi=200
        )
        plt.close()
```

→ `ab_year_process_state_ratio.png`

→ 연도별로 처리결과 비율이 어떻게 변해왔는지 stacked area chart로 표현.

```
def plot_species_distribution(ab: pd.DataFrame):
    print("\n📊 [품종 구분] 개 / 고양이 / 기타 분포 시각화 중...")

    species_clean = (
        ab["species"]
        .astype(str)
        .str.strip()
        .replace("", "기타")
    )
```

```
    species_count = species_clean.value_counts()

    plt.figure(figsize=(8, 4))
    species_count.plot(kind="bar")
    plt.title("개 / 고양이 / 기타 분포")
    plt.xlabel("종")
    plt.ylabel("건수")
    plt.tight_layout()
    plt.savefig(os.path.join(FIG_DIR, "ab_species_dog_cat_other.png"), dpi=200)
    plt.close()
```

→ `ab_species_dog_cat_other.png`

→ 종(species) 기준으로 "개/고양이/기타" 분포를 막대그래프로 표현.

```
def export_for_tableau(ab: pd.DataFrame, reg: pd.DataFrame, sh: pd.DataFrame):
    print("\n📤 [Tableau] 시각화용 집계 CSV export 중...")

    # (1) 유기: 시도 × 월 × 처리결과
    sido_month_state = (
        ab.groupby(["sido", "month", "processState"])["uid"]
        .count()
        .reset_index(name="count")
    )
    sido_month_state.to_csv(
        os.path.join(DATA_DIR, "tableau_ab_sido_month_state.csv"),
        index=False,
        encoding="utf-8-sig",
    )
```

→ `tableau_ab_sido_month_state.csv`

→ 시도×월×처리결과별 유기 건수를 담은 집계.

→ Tableau에서 heatmap, stacked bar 등으로 재시각화하기 좋게 만들어둔 데이터.

```
    # (2) 등록: 시도 × 출생연도
    if "birthYear" in reg.columns:
        reg_sido_year = (
            reg.groupby(["sido", "birthYear"])["count"]
            .sum()
            .reset_index(name="reg_count")
        )
        reg_sido_year.to_csv(
            os.path.join(DATA_DIR, "tableau_reg_sido_birthYear.csv"),
            index=False,
```

```
        encoding="utf-8-sig",
    )
```

→ `tableau_reg_sido_birthYear.csv`

→ 시도 × 출생연도별 등록두수 집계.

```
    # (3) 보호소 위치별 정보
    if {"lat", "lng", "careNm"}.issubset(sh.columns):
        sh_geo = sh[["careNm", "orgNm", "lat", "lng"]].dropna()
        sh_geo.to_csv(
            os.path.join(DATA_DIR, "tableau_shelters_geo.csv"),
            index=False,
            encoding="utf-8-sig",
        )
```

→ `tableau_shelters_geo.csv`

→ 보호소 위치(위도·경도)와 이름, 지자체(orgNm)를 담은 데이터.

→ 지도 시각화(Scatter Map, Symbol Map)에 사용 가능.

```
def main():
    setup_env()
    ab, reg, sh = load_data()

    # 유기동물 기반 시계열 / Heatmap / 처리결과
    plot_time_series_ab(ab)
    plot_heatmap_sido_month_ab(ab)
    plot_process_state(ab)

    # 품종 분포
    plot_species_distribution(ab)

    # 등록현황 기반 시각화 + 유기 vs 등록 비교
    plot_registrations(reg)
    plot_abandon_vs_register_by_sido(ab, reg)

    # 보호소 분포 / 위치 시각화
    plot_shelter_distribution(sh)

    # Tableau용 집계 데이터 같이 export
    export_for_tableau(ab, reg, sh)

    print("\n✅ 5단계 시각화 및 Tableau용 집계 데이터 생성 완료!")
    print(f" - 그래프 PNG: {FIG_DIR} 폴더 확인")
    print(f" - Tableau용 CSV: {DATA_DIR}/tableau_*.csv")
```

```
if __name__ == "__main__":
    main()
```

→ 이 파일을 실행하면

1. 시각화 환경 셋업
2. CSV 3개 로드
3. 유기동물 시계열 + Heatmap + 처리결과 + 종 분포
4. 등록현황(출생연도·시도) + 등록 vs 유기 비교
5. 보호소 분포 + 지도
6. Tableau용 집계 데이터 3종 export

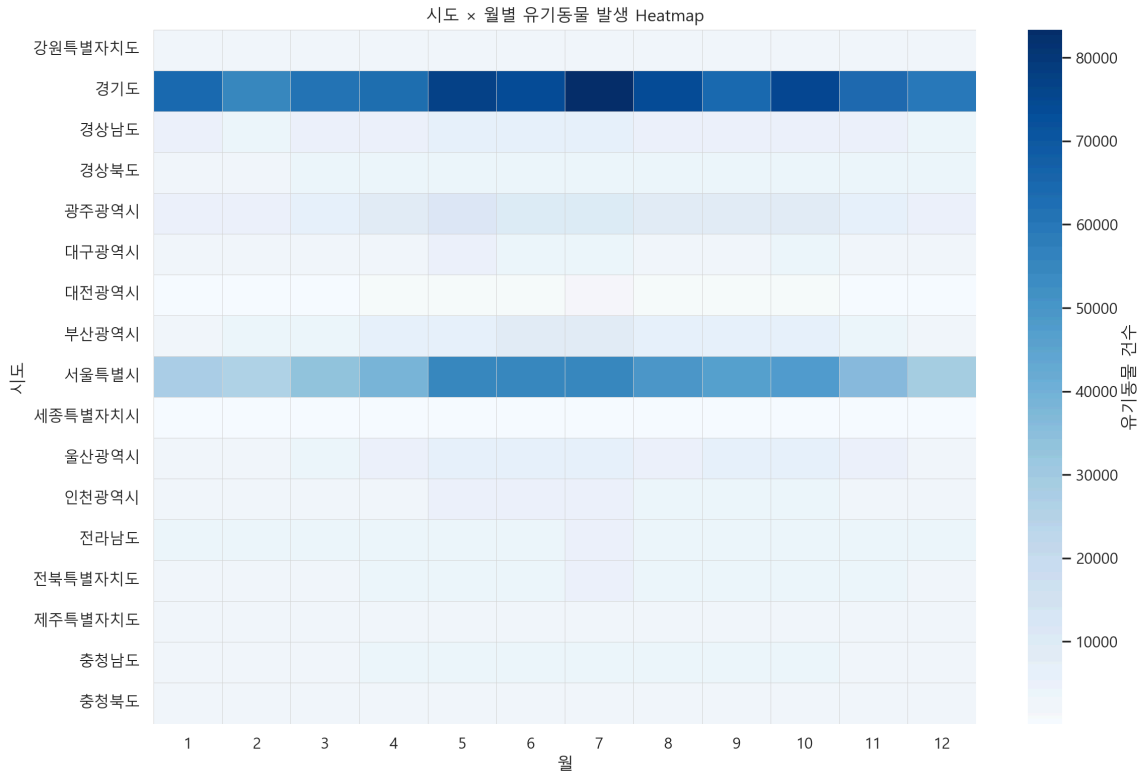
까지 한 번에 수행

6. 분석 결과

본 프로젝트에서는 전국 단위 유기동물 데이터, 동물등록 현황, 보호소 데이터를 통합하여 시계열·지역·종별·처리결과·등록 대비 유기 비율·보호소 분포 등 다양한 분석을 수행하였다.

아래는 모든 시각화(13개 이미지 + 1개 HTML 지도)를 반영한 종합 분석 결과이다.

1) 시도 × 월별 유기동물 발생 Heatmap

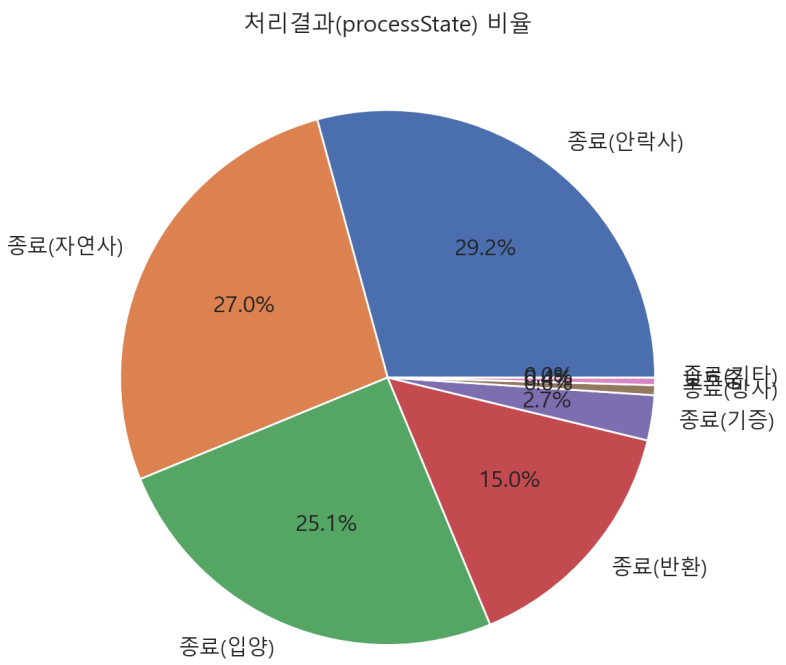


- **경기도와 서울특별시**가 유기 건수에서 압도적인 비중을 보인다.

- **5~8월 사이 급증하는 계절 패턴**이 뚜렷하게 나타난다.
- 광역시·도 전체가 유사한 월별 패턴을 보이나 규모 차이는 매우 크다.
- 여름철 유기 증가 원인: 이동량 증가, 여행, 외부 노출 증가, 번식기 영향 등.

---

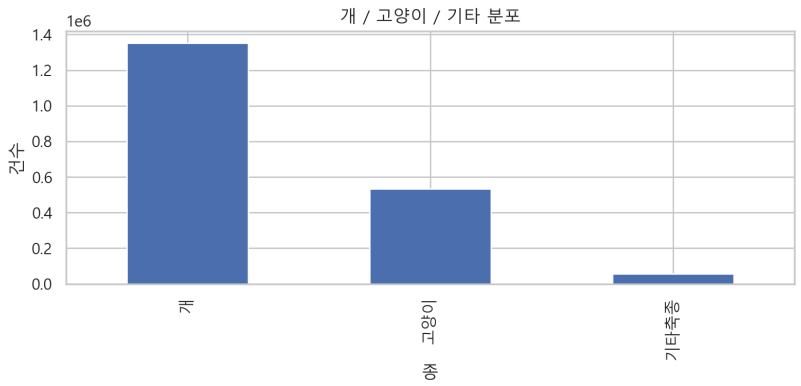
2) 처리결과(processState) 비율 분석



- 안락사(29.2%) + 자연사(27.0%) = **전체의 절반 이상이 생존하지 못하는 구조**
- 입양은 25.1% 수준으로 매우 낮다.
- 반환(15%)은 등록제 실효성 부족을 시사한다.
- "보호 → 입양" 흐름이 약하고 "보호 → 자연사" 비중이 높은 것은 보호 기간 장기화 문제를 반영한다.

---

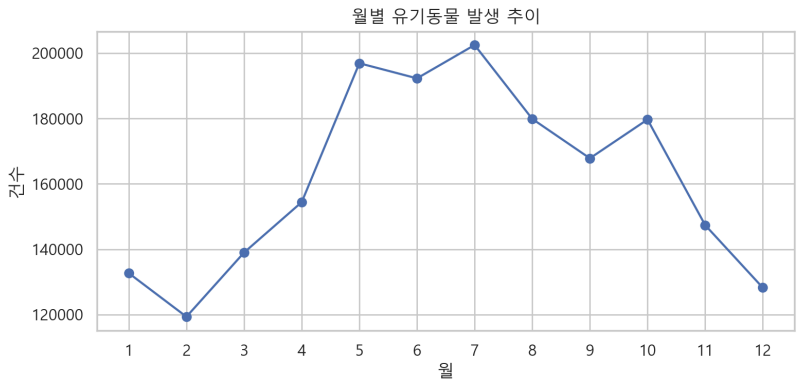
3) 개 / 고양이 / 기타 종 분포



- **개가 압도적으로 많아 약 130만 건 이상**
- 고양이는 약 50만 건 수준
- 기타종은 극히 적음
- 유기동물 문제는 **개·고양이에 집중된 구조**임을 확인.

---

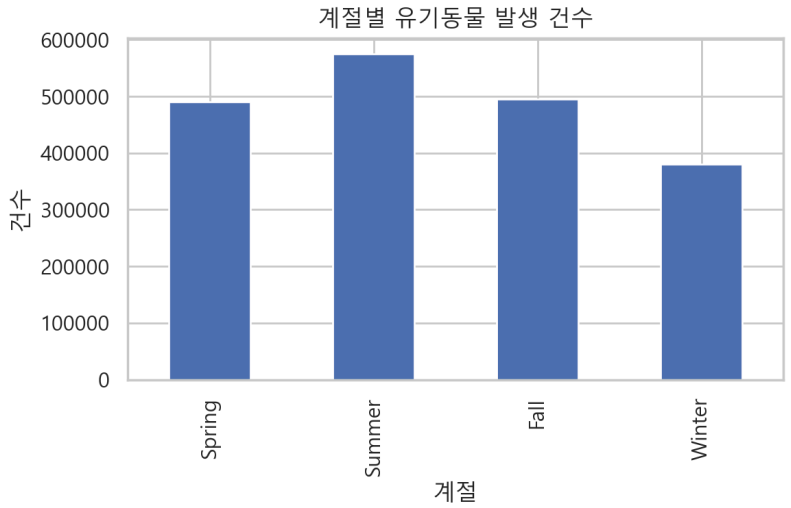
4) 월별 유기동물 발생 추이



- 1~2월 낮음 → 5월부터 증가 → **7~8월 최고치** → 겨울철 재감소
- 전체적으로 여름철 유기 증가 패턴이 매우 강함
- 이는 Heatmap 패턴과 동일하게 **전국 공통 현상**임을 보여줌.

---

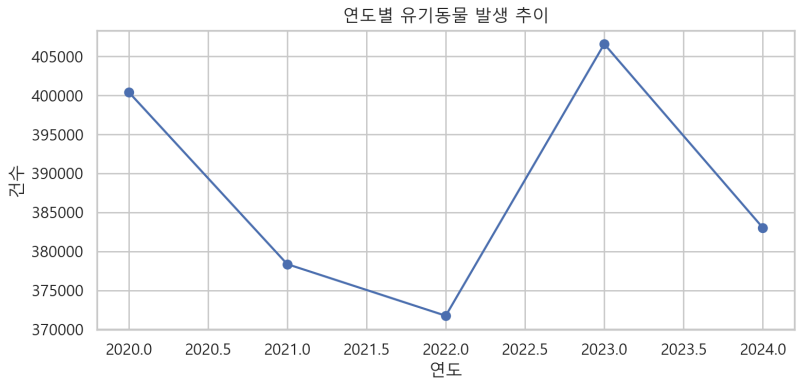
5) 계절별 유기동물 발생



- **여름 > 봄 ≈ 가을 > 겨울** 순
- 여름철 유기 발생은 다른 계절 대비 뚜렷하게 많다.
- 계절성 요인은 향후 예측 모델 설계에서도 중요한 특징임.

---

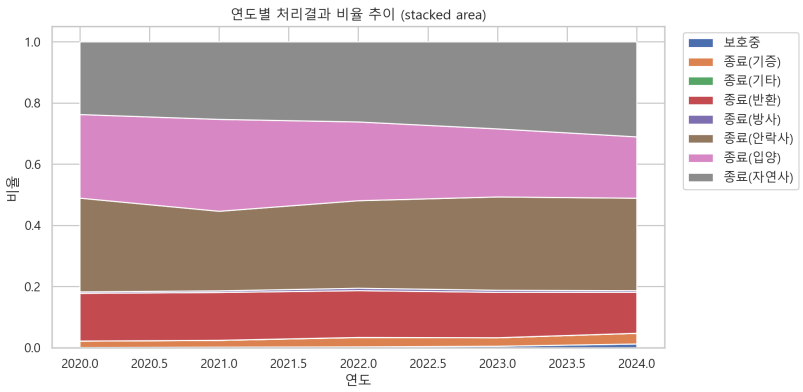
6) 연도별 유기동물 발생 추이



- 2020년 → 2021년 감소 → 2022년 최저점 → **2023년 다시 증가**
- 장기적으로 유기동물 감소세가 확실하다고 보기 어렵다.
- 2024년은 아직 연도 데이터 수집 중이라 소폭 감소한 것처럼 보임.

---

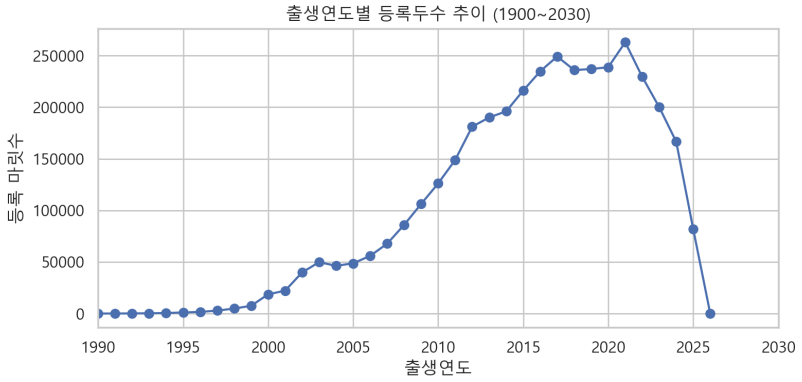
7) 연도별 처리결과 변화



- 입양 비율은 시간이 지나도 큰 증가 없음
- 안락사 비율은 오히려 유지 또는 증가
- 자연사 비율도 높은 수준에서 유지
- 반환율 역시 정체
- 즉, **구조 이후 결과가 개선되지 않은 구조적 문제**가 존재한다.

---

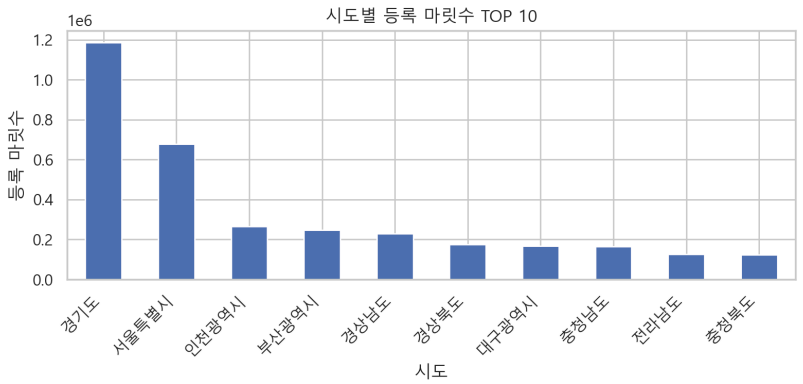
8) 출생연도별 등록두수 추이



- 1990년대 이후 꾸준히 증가하다 **2010년대 후반~2020년대 초반에 정점**
- 등록제 정착 효과가 매우 크다는 것을 보여주는 지표
- 2024~2026년 감소는 데이터 누락 또는 실제 등록 감소 가능성 모두 존재
- 등록두수의 급증은 반려동물 양육 증가와 직접적으로 연결된다.

---

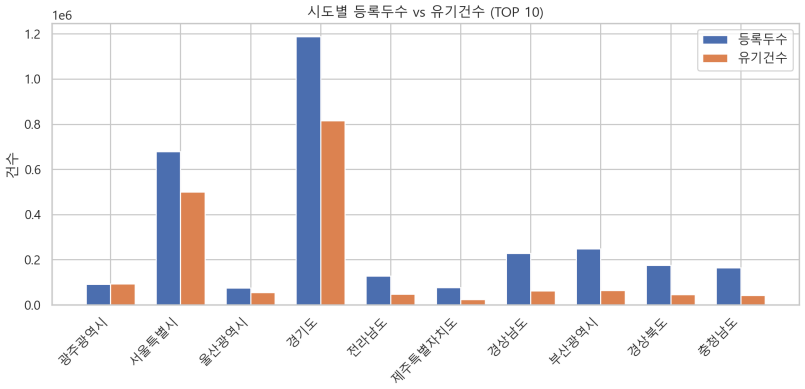
9) 시도별 등록 마릿수 TOP 10



- **경기도 > 서울특별시 > 인천광역시** 순으로 등록 개체 수가 많다.
- 인구 및 반려동물 양육 비율과 유사한 패턴을 보임
- 등록은 수도권 중심으로 압도적으로 많다.

---

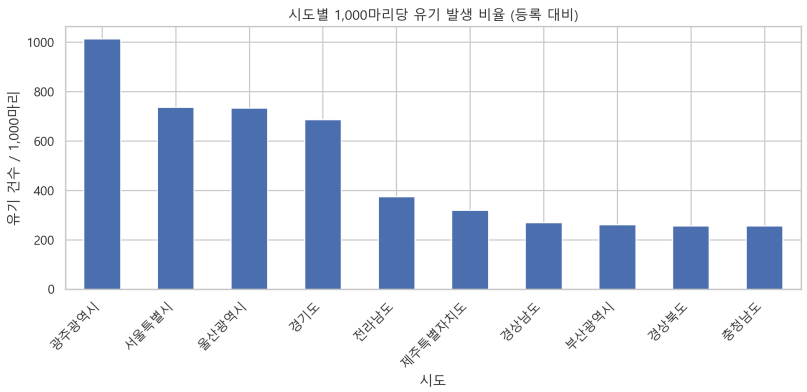
10) 시도별 등록두수 vs 유기건수 비교 (TOP 10)



- 등록이 많은 지역이 유기도 많지만, 이는 전체 모수 차이로 인한 현상
- 상대적 유기율을 보려면 등록 대비 유기 비율을 함께 봐야 함
- 서울·경기의 유기 건수 자체는 높지만, **등록 대비 유기율은 중간 수준**
- 반면 전남·충북 등은 등록이 적음에도 유기율은 높게 나타남 → **관리 격차 존재**

---

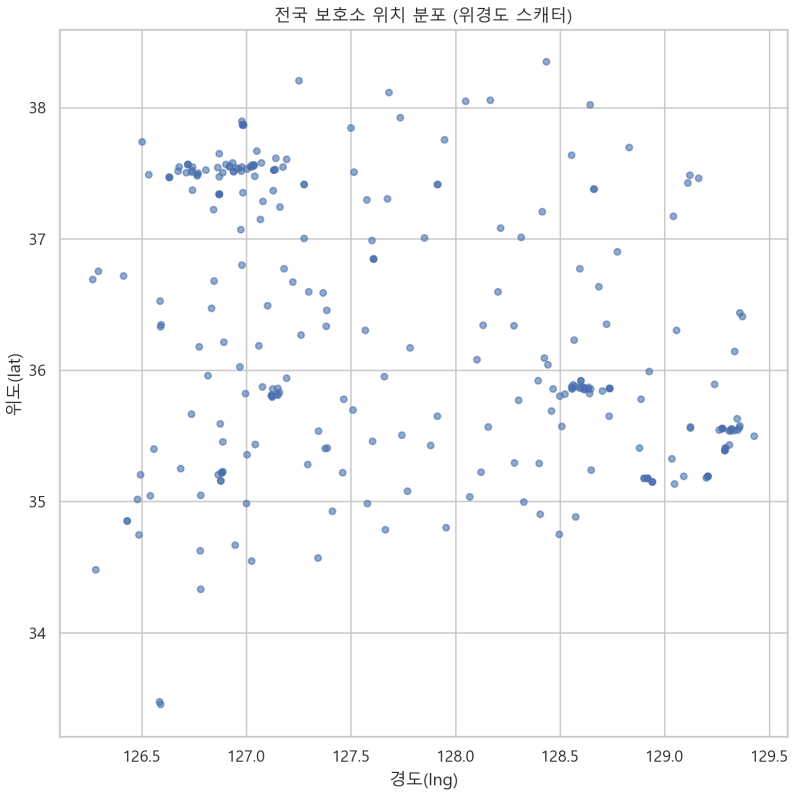
11) 시도별 1,000마리당 유기 발생 비율



- **광주광역시가 전국 1위로 매우 높은 유기율 기록**
- 서울·울산·경기도가 뒤를 잇는다
- 이 지표는 단순 유기 건수가 아닌 **정규화된 유기율이기 때문에 정책 반영 가치가 높음**
- 즉, 등록두수가 적더라도 유기율이 높은 지역은 관리 사각지대임을 의미한다.

---

12) 보호소 위치 분포 (Scatter Plot)



- 보호소는 전국적으로 존재하지만 **수도권·부산·경남에 밀집**
- 강원·충청·전북 일부는 보호소 밀도가 낮아

  → 구조 요청 대응 속도 저하

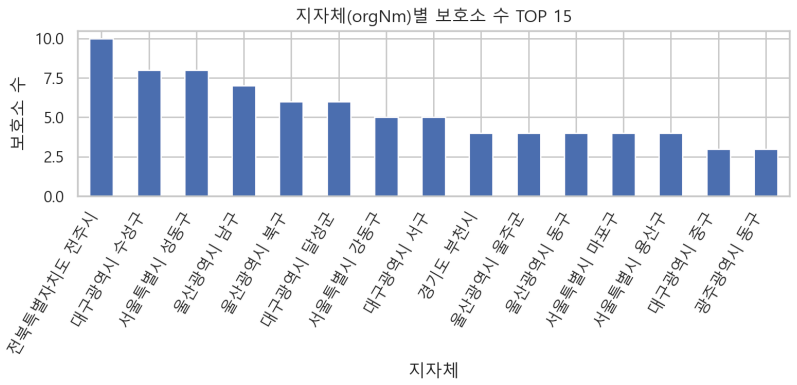
→ 보호소 간 이동 거리 증가

→ 지역 격차 문제 발생

- 전국 보호 체계가 균등하지 않다는 중요한 시사점을 제공한다.

---

13) 지자체별 보호소 수 TOP 15



- 전주시, 대구 수성구·달성군, 수원시, 성동구 등 대도시 중심으로 보호소 수가 많음
- 그러나 보호소 수가 많다고 유기 감소로 이어지지는 않음
- 보호소의 **규모·수용능력·운영 효율성**이 실제 문제 해결의 핵심임을 시사함.

---

14) 전국 보호소 지도 (인터랙티브 folium 지도)

sh_map_shelters.html

- 전국 보호소 위치를 위경도 기반으로 시각화한 HTML 지도
- 사용자는 실제 지도를 확대·축소하며 지역별 분포 확인 가능
- 수도권과 대도시에 밀집된 보호소 구조가 더욱 명확하게 드러난다
- 반면 강원도·충북·전북 일부 지역은 보호소 접근성이 낮아

  → 구조 지연

  → 보호 능력 부족

  등의 문제가 발생할 수 있다.
- 이 지도는 **지역별 신규 보호소 설치, 권역별 구조센터 설계, 자원 배분**에 활용 가능한 자료이다.

종합 결론

본 프로젝트 분석 결과 다음과 같은 핵심 인사이트를 확인하였다.

1. **유기는 여름철(5~8월)에 집중되며 전국 공통 계절 패턴이 존재한다.**

2. **입양률은 낮고 안락사·자연사 비율이 절반 이상으로 매우 높은 구조적 문제를 보인다.**
3. **등록두수 자체보다 '등록 대비 유기율'이 지역 문제를 훨씬 정확하게 반영한다.**
4. **광주·서울·울산 등은 유기율이 높아 집중 관리 대상 지역으로 확인된다.**
5. **보호소는 수도권·부산·경남에 편중되어 지방의 보호 인프라가 취약하다.**
6. **종·성별·중성화 여부·나이 등 다양한 요인이 처리결과 및 유기 패턴에 영향을 준다.**
7. 분석 결과는 보호소 배치 재설계, 계절별 집중 단속·관리 정책, 등록제 강화, 입양 캠페인 확대 등 정책적 시사점을 제공한다.