ChatGPT

네이버 쇼핑 광고 크롤러 로직 문제 분석 및 개선 방안

문제 현황 및 원인 분석

현재 Puppeteer를 활용한 네이버 쇼핑 **상품형 광고** 크롤링에서 기술적으로는 브라우저 실행과 페이지 접근까지 성공했지만, **실제 콘텐츠가 로드되지 않는 문제**가 발생하고 있습니다. 로그를 보면 **상품 리스트의 앵커(`anchors`) 개수가 0개**이고 광고 카드(`adCards`)도 0개로 검출되어, **검색 결과가 전혀 표시되지 않는 상태**입니다. 이는 다음과 같은 원인으로 추정됩니다:

- **자바스크립트 동적 콘텐츠 미수집:** 네이버 쇼핑 검색 결과는 클라이언트 측 **자바스크립트로 동적으로 렌더링**되는 부분이 많습니다. 현재 코드는 `page.goto(..., waitUntil: "domcontentloaded")` 후 **고정 3초 대기**만 하고 있어, **JS로 로드된 상품 리스트를 기다리지 못한 가능성**이 있습니다. 실제로 네이버의 페이지 구조상 초기 HTML에는 뼈대만 있고, **제품 목록과 광고는 추가 AJAX 호출로 가져와 렌더링**될 수 있습니다 [1]. 이러한 **포스트-로드(post-load) 데이터**를 제대로 감지하지 못하면 **데이터가 빈 채로 크롤링되어 정확도가 떨어집니다** [1].

- **반(反)봇 감지로 인한 콘텐츠 차단:** 로그에서 5페이지를 연속 스캔했음에도 결과가 0인 점으로 미루어, 네이버 측에서 **자동화 접근을 탐지하여 콘텐츠 로딩을 아예 막았을 가능성**이 높습니다. 네이버는 **비정상적 패턴이나 봇 행동을 적극 탐지**하며, 특정 **세션이 의심스러우면 검색 결과 대신 빈 페이지를 보여주거나 CAPTCHA를 요구**할 수 있습니다 [2] [3]. 현재 코드에서 스텔스 플러그인을 제거하고 **고정된 User-Agent (`Chrome/123.0.0.0`)** 등을 사용한 점, 그리고 **짧은 간격으로 다수 페이지 요청**한 점 등이 **봇으로 간주될 요소**입니다. 네이버의 **안티봇 시스템은 IP뿐만 아니라 행동 패턴까지 모니터링**하므로 [3], **빠르고 동일한 간격**의 페이지 이동, **헤드리스 브라우저 티가 나는 환경** 등이 겹쳐 **컨텐츠 비노출(차단)**로 이어진 것으로 보입니다.

- **광고 디스플레이 방식 변화 가능성:** 혹은 해당 키워드에 **실제로 해당 상품 광고가 없거나**, 네이버 쇼핑의 **광고 표시 구조가 바뀌어** 기존 방식으로 광고 요소를 찾지 못했을 수도 있습니다. 하지만 **일반 상품 앵커까지 0개**인 것은 구조 변화보다는 앞선 **콘텐츠 미렌더/차단** 가능성에 무게를 실어줍니다.

위 문제들로 인해 **크롤러의 정확성(정확히 광고를 찾아내는 능력)**과 **안정성(항상 결과를 얻는 능력)**이 매우 저하된 상태입니다. 따라서 **정확성을 최우선**으로, 그 다음으로 **안정적인 검색과 탐지 우회**를 도모할 수 있도록 **크롤링 로직의 전면 개선**이 필요합니다.

개선 방안 개요

개선을 위해 **"정확성 → 안정성 → 탐지 우회"** 순으로 우선순위를 두고 로직을 재구성합니다. 특히 **상품형 광고만 대상으로** 하므로, 해당 광고를 정확히 식별하고 차단 없이 가져오는 데 초점을 맞춥니다. Puppeteer 사용에는 제한이 없으므로 (헤드리스/헤드풀 등 모두 사용 가능), **사람처럼 보이는 Puppeteer 설정**과 **충분한 대기/검증 로직**을 적용해 신뢰성을 높이겠습니다. 아래에 구체적인 개선 전략을 설명합니다.

1. 자바스크립트 동적 콘텐츠 로드 대기 (정확성 향상)

정확성을 높이기 위해서는 **페이지의 동적 콘텐츠가 완전히 로드된 후** 광고를 검색해야 합니다. 이를 위해 아래 조치를 취합니다:

- **네트워크 및 DOM 상태를 이용한 대기:** `waitUntil: "networkidle2"` 옵션으로 페이지 이동 시 **네트워크 요청이 완전히 안정될 때까지 대기**하거나, 특정 **요소가 나타날 때까지 기다리는** 방식으로 변경합니다. 예를 들어 검색 결과 목록의 첫번째 상품 카드 요소 또는 **제품 앵커(**`a[href*="nvMid="]`**)**가 나타날 때까지 `page.waitForSelector`를 사용합니다. 이렇게 하면 **JS가 추가 로딩하는 상품 목록이 표시될 때까지** 충분히 기다릴 수 있습니다. 또한 고정 3초 대기 대신 **동적으로 조건을 검사하며 대기**함으로써 정확도를 높입니다[1]. 만약 지정 시간 내에 요소가 안 나타나면 탐지 차단을 의심할 수 있으므로 그에 대한 별도 처리를 합니다.

- **스크롤 등 사용자 동작 시뮬레이션:** 필요하다면 **페이지의 스크롤을 인위적으로 발생**시켜 **지연 로드(lazy-load)되는 콘텐츠나 광고**를 불러옵니다. 네이버는 **게으른 로딩과 스크롤 트리거로 사용자 행동을 모방**하는 경우가 있어[2], 첫 페이지 로드 후 `window.scrollBy()`나 `page.mouse.wheel()` 등을 활용해 **천천히 페이지 끝까지 스크롤**하고, 해당 동작 후 잠시 대기하여 **숨겨진 광고 콘텐츠도 로드**되도록 유도합니다. 이러한 **인간적인 스크롤 인터랙션**은 **숨겨진 콘텐츠를 불러옴과 동시에, 봇 탐지 회피 효과**도 일부 얻을 수 있습니다[2].

- **광고 식별 로직 보강:** 광고를 찾는 DOM 탐색 로직도 **네이버 최신 구조에 맞게 재검토**합니다. 현재는 카드 내 텍스트에 `"광고"`, `"AD"`, `"스폰서"` 등의 단어가 있는지로 광고 여부를 판단하고 있는데, 네이버 쇼핑의 상품형 광고에는 **"광고" 라벨이 아이콘이나 별도 요소로 존재**할 수 있습니다. 따라서 **광고 배지를 나타내는 요소**(예: 클래스명에 `__ad-badge` 또는 스크린 리더용 `"광고"` 텍스트가 있는 `span`)를 찾거나, **네이버 쇼핑 광고 특유의 DOM 구조**(예: 광고 영역은 리스트 상단부에 별도 `<ul>`에 묶인다든지, `data-expose=\"advertisement\"` 같은 속성이 있을 수 있음)를 활용하는 것이 좋습니다. 구조 정보를 파악하기 위해 브라우저 개발자도구로 실제 검색결과를 확인하고, **가장 신뢰도 높은 선택자**를 사용하도록 코드 수정이 필요합니다. 이렇게 하면 광고 탐지 **정확도를 향상**시킬 수 있습니다.

- **데이터 추출 정확성:** 광고를 찾았을 때 **스토어명, 가격 등 부가정보 추출 로직**도 점검합니다. 현재는 클래스명에 `"mall"`, `"store"` 등이 들어간 요소를 찾아 상점명을 얻고 `"price"`가 포함된 클래스의 텍스트에서 숫자를 추출하는데, 해당 클래스명이 바뀌거나 여러 개일 수 있습니다. **정확성 향상**을 위해 **특정 하나의 selector로 상점명과 가격을 타겟팅**하는 것이 좋습니다 (예: `card.querySelector('.productInfo__seller .seller_name')` 등 실제 구조에 맞는 선택자). 사이트 구조 변경에 대비해 **예외 처리를 추가**하고, 만약 해당 요소를 못 찾으면 전체 카드 텍스트에서 패턴 매칭하는 backup 방식으로 **이중 체크**를 할 수도 있습니다.

2. 봇 탐지 우회 및 세션 전략 (안정성 향상)

네이버의 **반봇 시스템을 우회**하여 안정적으로 데이터를 얻으려면, **사람과 유사한 세션 환경**을 조성해야 합니다. 다음과 같은 대책을 적용합니다:

- **Puppeteer Stealth 플러그인 사용:** 이전 코드에서 제거했던 `puppeteer-extra-plugin-stealth`를 다시 활용합니다. Stealth 플러그인은 헤드리스 크롬과 실제 사용자 브라우저의 **미세한 차이를 보정**해 주며, `navigator.webdriver` 등의 속성을 숨겨주고 WebGL, 헤드리스 User-Agent 문자열 등을 실제 브라우저처럼 꾸며줍니다[4]. Stealth 플러그인을 `puppeteer.use()`로 적용하고 기본 설정을 켜두면, **헤드리스 브라우저 표시를 최대한 없애** 웹사이트의 봇 탐지가 어려워집니다[4].
  ※ **참고:** Stealth 적용 시에도 현재 수동으로 설정한 UA, plugins, languages 등이 중복으로 적용되지 않도록 정리합니다. 가능하면 **실제 사용자 UA 문자열**(예: 최신 Chrome 버전)로 설정하고, `page.setExtraHTTPHeaders`로 **Accept-Language, Accept 등** 일반 브라우저와 동일한 헤더를 포함시키는 것이 권장됩니다[5][6].

- **지나치게 빠른 탐색 자제: 인간의 행동 패턴에 맞춰 요청 간격을 랜덤 지연**시킵니다. 현재 1~1.7초 정도의 페이지 간 딜레이를 주었지만, 이것을 **5~10초 사이의 랜덤 지연**으로 늘리고 첫 페이지 로드 후에도 **일정 시간 (예: 5~15초 랜덤) 머무르도록** 합니다. 실제 사용자들은 검색 결과를 보고 **바로 1초만에 페이지를 넘기지 않으므로**, 이러한 **지연 전략으로 비정상적인 속도를 피해야** 합니다 [7]. 또한 한 세션(브라우저 인스턴스)에서 너무 많은 페이지를 연달아 크롤링하지 않고, **몇 페이지마다 새로운 브라우저(또는 새 탭)**를 여는 것도 방법입니다. 네이버는 **짧은 시간 다량의 요청이나 패턴화된 행동을 탐지**하므로 [3], 요청 수와 속도를 **사람 수준으로 제한**하는 것이 중요합니다 [8].

- **프록시 및 지역 IP 활용:** 만약 서버가 해외 IP이거나 동일 IP로 반복 조회 시 차단될 경우를 대비해, **로테이팅 프록시**나 **국내 IP 프록시**를 사용하는 것도 고려합니다. 현재 함수에 `proxy` 옵션을 넣어둔 것은 이러한 대비책으로 보이는데, 필요 시 **IP를 교체**하여 접근하면 **지역 및 세션 다양성**을 높일 수 있습니다 [3]. 다만 **네이버는 IP보다도 행동 패턴을 중시하여 차단**하는 경향이 있으므로 [3], 프록시를 쓰더라도 **동일한 봇 패턴**이면 효과가 제한적입니다. 따라서 프록시는 **보조 수단**으로 활용하고, 근본적으로는 **세션 쿠키 유지(로그인 상태 활용 등)**나 **클릭/이동 등 사용자 흐름 재현**이 더 중요할 수 있습니다 [9]. 가능하다면 **로그인 세션을 유지한 채** 검색하거나, **쿠키/로컬스토리지**를 한 번 받아서 재사용하는 방식도 추후 안정성에 기여할 수 있습니다.

- **에러 및 우회 처리:** 안정성을 위해 **예외 상황별 처리 로직**을 추가합니다. 예를 들어, **콘텐츠가 끝까지 안 나타나는 경우** (여전히 anchors=0인 경우)에는 해당 세션이 차단된 것으로 보고 **브라우저를 새로 열어 재시도**하거나, 일정 시간 **대기 후 재시도**합니다. 또한 네이버가 **CAPTCHA 페이지**를 응답하는 경우를 대비해, 페이지 내용에 `"보안을 위해 확인"` 등의 문구나 CAPTCHA 이미지 태그가 있는지 검사하여 감지하면, 이를 만났을 때는 **즉시 실패를 알리거나** (혹은 캡차 솔빙 절차를 넣어야 하지만 이는 범위 밖) **다른 세션으로 우회**하는 전략을 취합니다. 이런 식으로 **실패 케이스별 fallback 경로**를 마련해 두면 크롤러가 갑자기 죽지 않고, **최소한의 안내나 재시도**를 수행하여 안정적으로 동작할 수 있습니다.

3. 로직 재구성 요약 및 예상 흐름

위 개선사항을 토대로, `fetchAdRank` 함수의 동작 흐름을 전면 재구성하면 다음과 같습니다:

1. **브라우저 초기화 및 페이지 설정 강화:** `puppeteer-extra`와 Stealth 플러그인을 사용하여 브라우저 실행. `headless: false` (디버깅 시) 또는 `"new"` 모드 사용. **실제 Chrome 브라우저와 최대한 동일한 UA 문자열과 헤더** 세팅 [5]. 필요 시 브라우저 창 크기, timezone, locale 등도 실제 사용자 환경과 일치시킵니다. (참고: Stealth가 대부분 처리하지만, UA는 수동 설정 추천.)

2. **검색 페이지 접속 (1페이지):** `page.goto` 호출 시 `waitUntil: "networkidle2"` 옵션으로 대부분의 네트워크 활동이 끝날 때까지 대기. 이어서 **페이지 핵심 요소 대기**: 예를 들어 `await page.waitForSelector('.list_basis', { timeout: 5000 })` (가상 예시로, 검색 결과 리스트의 컨테이너 클래스) 등을 사용해 **결과 목록이 DOM에 추가될 때까지 기다림**. 만약 해당 시간 내 나타나지 않으면 `page.content()`를 출력/검사하여 **차단 징후**(캡차 등)를 로그로 남기고 우회 절차로 넘어갑니다.

3. **콘텐츠 로드 및 사용자 행동 시뮬레이션:** 페이지가 열리면 **인위적 딜레이 및 인터랙션**을 추가합니다. 예를 들어 첫 페이지 결과가 떠 있더라도 **1~3초 정도 자연 대기**하고, `page.mouse.move()`를 사용해 화면 내 임의 지점을 살짝 이동하거나, `page.mouse.wheel({ deltaY: 500 })` 등으로 **천천히 스크롤**합니다. 그런 다음 추가로 **상품 앵커들이 모두 로드되었는지** (`document.querySelectorAll('a[href*="nvMid="]')` 개수 체크 등) 확인하고, 부족하면 약간 더 대기합니다. 이 과정에서 **불필요한 리소스 (이미지 등)는 로딩 차단**도 고려할 수 있으나 [10], 이미지 로딩이 봇 여부 판단에 쓰일 수도 있으므로 우선은 **모두 로드**하는 편이 안전합니다.

4. **광고 아이템 스캔 (정확한 식별):** 로드된 페이지에서 **광고 항목만 필터링**합니다. 개선된 로직은 다음과 같습니다:

5. 모든 상품 **카드 요소**를 가져옵니다 (예: `document.querySelectorAll('ul.list_basis > li')` 와 같이 구체적으로 리스트 아이템을 선택).
6. 각 카드에서 **"광고" 배지 유무**를 검사합니다. 예를 들어 `card.querySelector('.ad_advertise')` 요소가 존재하거나, `card.innerText` 또는 접근성 속성에 "광고"가 포함되면 **해당 카드가 광고**임을 확정합니다.
7. 광고임이 확인된 카드들에 대해, 그 내부에서 **상품 ID (nvMid)**를 포함하는 링크를 찾습니다. `a[href*="nvMid="]` 혹은 정규식으로 `nvMid=(\d+)` 추출. 이때 **타겟 productId와 일치**하는지 검사합니다. (여기서 문자열 비교 시 불필요한 선행 `0` 제거 등 현재 `eqNumStr` 함수 로직 유지.)

8. 매칭되면 해당 카드의 **순서 (광고 내 순위)**를 파악합니다. 광고 목록 내 index를 이용하거나, 페이지 내 전체 광고 중 몇 번째인지 추적합니다. 현재처럼 `cumulativeAdCount`로 전 페이지까지의 광고 수를 누적하고 현재 페이지의 광고 index를 더해 **전체 광고 순위**를 구하는 방법을 그대로 쓰되, **광고 카운트 집계가 정확히 이뤄지도록** 합니다 (예: 혹시 광고를 건너뛰고 카운트하는 로직상의 버그가 없는지 점검).

9. **다음 페이지 이동:** 현 페이지에서 제품을 찾지 못했고, `pageIndex < maxPages`인 경우 **다음 페이지로 이동**합니다. 이동 방법은 두 가지 옵션:

10. **링크 클릭:** 실제 사용자처럼 **"2페이지" 버튼을 클릭**하여 페이지 전환. 이를 위해 `await page.click('a.pagination_btn_next')` 또는 페이지 번호 링크를 클릭하는 방식을 활용합니다. 이렇게 하면 **내부 SPA 네비게이션이나 히스토리 API를 통해 전환**될 수 있어, 더 자연스러운 이동이 됩니다. 클릭 후에도 **위와 동일하게 콘텐츠 로드 대기 및 광고 스캔**을 수행합니다.

11. **URL 직접 변경:** 또는 현재처럼 URL의 `pagingIndex` 파라미터만 바꿔 `page.goto`로 이동할 수도 있습니다. 이 경우도 마찬가지로 **각 페이지마다 충분한 대기와 인간적 지연**을 적용합니다. 다만 연속 `goto` 호출은 **초당 여러 페이지 요청으로 인지**될 수 있으므로, **페이지 간격을 더 늘리고** 콘솔에 로그도 최소화하는 등 자연스러움을 모방합니다.

12. **결과 반환 및 예외 처리:** 목표 productId를 발견하면 즉시 `return`하여 결과를 반환합니다 (발견한 페이지, 광고 순위, 가격 등). 만약 `maxPages`까지 찾아도 없으면 `found: false`와 함께 **"광고 결과에 미노출"** 등의 메시지를 포함해 반환합니다. **추가로**, 크롤링 중간에 **에러나 탐지 의심 상황**(예: 페이지 내용이 아예 없거나 오류 발생)이 감지되면, 해당 오류 내용을 `notes`에 남겨 디버깅에 활용합니다. 예를 들어 "네이버 봇 감지로 콘텐츠 미노출 추정, n회 재시도 실패"와 같은 노트를 남겨 개발자가 후속 조치를 판단할 수 있도록 합니다.

以上의 흐름으로 로직을 개선하면, **정확성** 면에서는 동적 콘텐츠까지 모두 포함하여 광고를 찾을 수 있고, **안정성** 면에서는 네이버의 탐지에 걸리지 않고 지속적으로 결과를 얻을 가능성이 높아집니다. 실제 **네이버의 반봇 시스템**은 매우 정교하므로 [2] [3], 지속적으로 **사람처럼 세션 행동**을 흉내내고 **사이트 구조 변경에 따라 코드도 업데이트**하는 노력이 필요합니다. 하지만 제안한 대로 **Stealth 적용, 딜레이 조정, 사용자 행동 시뮬레이션** 등을 도입하면 **크롤러가 훨씬 안정적으로 광고를 포착**할 수 있을 것입니다 [7] [3].

추가 고려사항

- **지속적인 모니터링 및 유지보수:** 네이버는 **레이아웃이나 구조를 수시로 변경**할 수 있습니다 [11]. 따라서 크롤러가 한동안 잘 동작하더라도, **정기적으로 결과 정확도를 모니터링**하고 **필요시 셀렉터나 로직을 업데이트**해야 합니다. 또한 **네이버의 차단 상황**(예: 특정 IP에서 반복 접근 시 캡차 발생)을 발견하면, **IP 교체나 요청 간격 추가 조정** 등의 대응을 꾸준히 해나가야 합니다.

- **윤리 및 이용 약관 준수:** 기술적으로 우회를 하더라도 **네이버 서비스 이용약관 및 로봇 배제규약(robots.txt)**을 준수하는 것이 중요합니다. 대량의 데이터 수집은 서비스에 부담을 줄 수 있으므로 **과도한 병렬 요청은 피하고 속도를 조절**해야 합니다 [12]. 또한 수집한 데이터를 사용하는 목적이 네이버 정책에 어긋나지 않는지 유의해야 합니다.

- **로그인 활용 가능성:** 상품형 광고는 로그인 없이도 보이지만, **로그인한 사용자에게만 보이는 콘텐츠**가 있을 수 있고, 로그인하면 오히려 **추가적인 봇 검증 절차**가 적용될 수도 있습니다. 현재는 비로그인 상태로 크롤링하고 있으므로, **향후 필요에 따라 네이버 계정 세션 쿠키를 적용**해 볼 수 있습니다. 다만 이 경우 **로그인 시도의 자동화**는 더 엄격한 탐지 대상이 될 수 있어 주의가 필요합니다.

以上의 개선안을 면밀히 구현하고 테스트하면, **"정확성 최우선, 안정적 검색, 탐지 우회"**라는 목표에 부합하는 크롤링 모듈을 구축할 수 있을 것으로 기대됩니다. 네이버의 반자동화 전략에 대응하여 **세션 행동을 인간에 맞추고** [9] **, 우회보다** 모방**에 중점** [13] 을 두면 장기적으로 더 안정적인 데이터 수집이 가능할 것입니다.

참고자료:

- ZenRows 블로그, "How to Avoid Detection with Puppeteer", Puppeteer 스텔스 및 탐지 회피 기법 소개 [4] [7]
- GroupBWT 기술 블로그, "Scraping Naver Platform: ..." (2025), 네이버의 반봇 감지 전략과 시사점 [2] [3] [1]
- RealDataAPI 가이드, "Naver Shopping Product Scraper", 네이버 쇼핑 데이터 수집 개요 (동적 콘텐츠 처리 중요성 언급) [1] (내용 참고)

---

[1] [2] [3] [9] [11] [12] [13] How to Scrape Data from Naver for Strategic Data Extraction in 2025
https://groupbwt.com/blog/how-to-scrape-data-from-naver/

[4] [5] [6] [7] [8] [10] How to Avoid Detection with Puppeteer | by ZenRows | Medium
https://medium.com/@zenrows/puppeteer-avoid-detection-517a252eb27