

데이터' 없는 인공지능(AI)은 무용지물이라는 말, 들어 보셨나요? AI 는 데이터 학습에서부터 시작되고 데이터의 양과 질이 AI 의 성능을 좌우할 만큼, AI 에서 데이터는 중요한 역할을 하는데요. 지난 9 일 데이터의 효율적인 활용을 위해 추진된 '데이터 3 법(개인정보보호법, 신용정보법, 정보통신망법)'이 국회 본회의를 통과하면서 관련 산업 발전에 대한 기대가 커지기도 했죠.

이스트소프트 또한 2016 년부터 AI 신사업을 전개하면서 AI 데이터의 중요성을 여실히 느꼈습니다. AI 앱 개발에 필요한 데이터를 지속적으로 수집하고, 필요한 형태로 가공하는 일이 쉽지 않았기 때문입니다. 이러한 이유로, AI 연구소가 설립된 지 1 년만에 데이터 구축 업무를 전담하는 '데이터인텔리전스팀'을 신설하여 전문인력을 양성하고 있습니다.

실제로 AI 분야의 글로벌 리더인 미국, 중국에는 이미 유사한 Data Collection / Annotation Service 를 주요 비즈니스 모델로 하여 성장한 기업들이 꽤 있습니다. **Scale AI** 는 2016 년 미국에서 22 살의 MIT 졸업생이 창업한 데이터 구축 전문 스타트업인데요. 지난 8 월에 1 억 달러의 series C 펀딩을 완료하여 총 10 억 달러(약 1 조 2 천억 원)의 기업 가치를 달성했습니다. Grand View Research 에 따르면 2025 년에는 전 세계 데이터 어노테이션 툴의 시장 규모만 해도 16 억 달러(약 1 조 9 천억원)에 이를 것이라고 합니다.

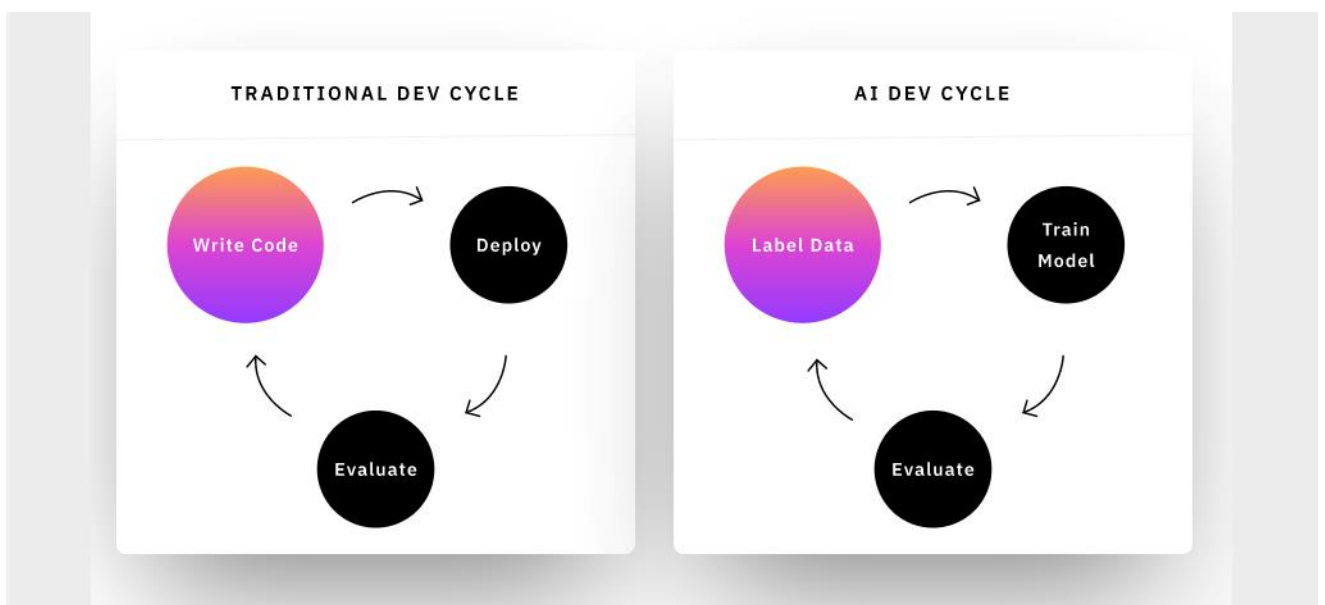


그림1. The AI development cycle depends on labeling high-quality data. (Source: Scale AI)

앞서 '메트릭러닝 기반 안경 검색 서비스 개발기'에서도 '데이터인텔리전스팀과의 협업으로 데이터셋을 구축했다'는 내용이 언급된 적이 있었는데요. 이번 글에서는 반려동물 전용 갤러리 앱 '**포에버(PAWEVER)**' 프로젝트 사례를 통해, 우리가 어떻게 AI 데이터를 수집하고, 가공하고, 활용하는지에 초점을 맞춰 이스트소프트의 AI 데이터 구축 과정 전반을 소개하려 합니다.

포에버(PAWEVER)의 기획

데이터 이야기를 하기에 앞서, 포에버(PAWEVER)에 대한 간단한 소개를 하겠습니다. 앱의 특성과 기획의도를 먼저 알아야 데이터 구축 과정에 대해 보다 쉽게 이해할 수 있기 때문입니다.

이스트소프트는 '실용주의 인공지능'을 지향하며 AI 기술이 보다 쉽고 재미있게 사용자들의 삶에 스며들 수 있는 서비스를 늘 고민하고 있습니다. 그 시작은 딥러닝 기술로 이미지에서 하늘 영역을 인지하여 원하는 하늘로 바꿔주는 카메라앱 '**피크닉**'입니다. 대대적인 마케팅 투자 없이도 글로벌 1,000 만 다운로드를 돌파한 이스트소프트의 효자 앱이죠.

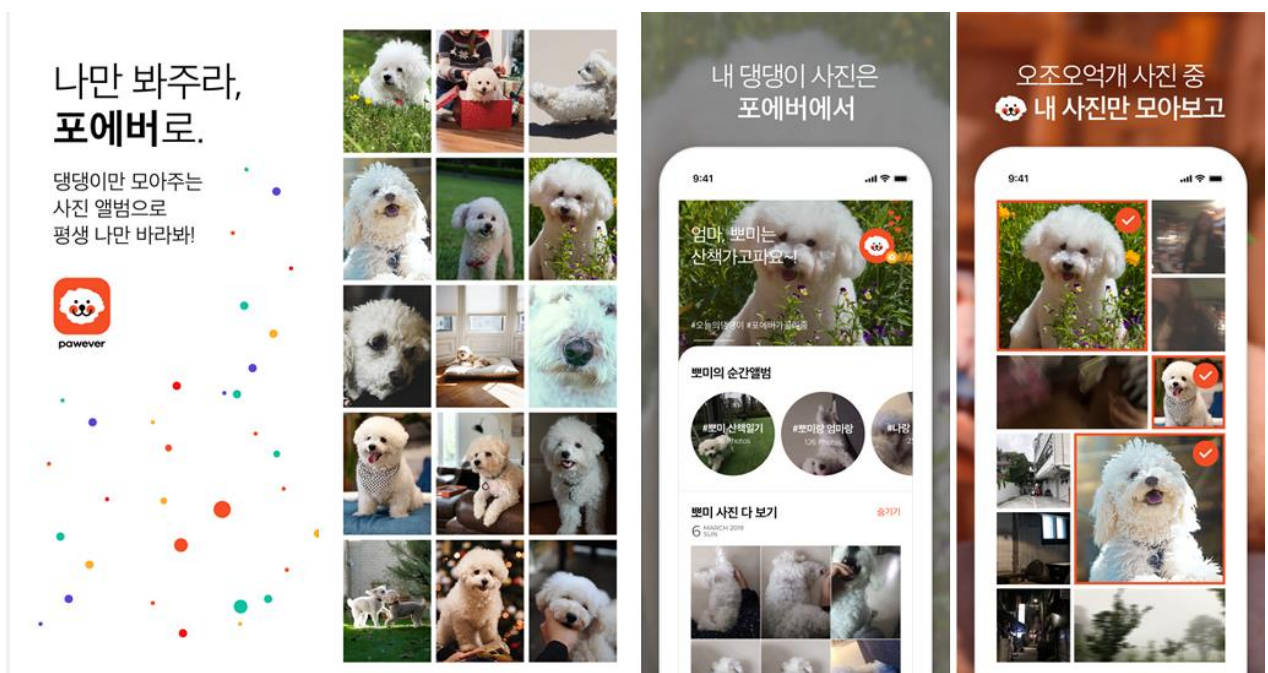


그림2. 포에버(PAWEVER) 앱

피크닉이 승승장구하고 있을 때, 우리는 다시 우리가 가진 AI 기술을 가지고 어떤 새로운 서비스를 만들 수 있을지 치열하게 고민했고, 포에버(PAWEVER)를 기획하게 됩니다. **포에버(PAWEVER)는 내 스마트폰 사진첩의 다양한 사진들 중 강아지 사진만을 모아 보여주는 갤러리앱**입니다. 포에버(PAWEVER) 속 AI 기술은 많고 많은 사진 중에 우리 반려견의 얼굴을 인지하고, 분류해낼 수 있는 능력을 가진 것인데요. 이러한 AI 모델과 포에버(PAWEVER) 탄생의 이면에는 여러 단계에 걸친 방대한 데이터의 구축 과정이 있었습니다.

포에버(PAWEVER) 앱 개발을 위한 AI 데이터 구축

(1) 훈련 데이터셋 수집(Collection of Training Data set)

포에버(PAWEVER)의 기획을 실현시키기 위해서는 우리 AI 연구소에서 보유한 AI 모델 중 '**분류모델**'을 활용해야 했습니다. 말 그대로 다양한 사진 속에서 반려견의 사진만 '**분류**'해내는 것입니다. 분류모델이 반려견의 이미지를 잘 구분해낼 수 있도록 학습시키려면 다양한 카테고리의 반려견 이미지를 수집하는 것이 최우선입니다. AI 모델을 '**훈련**'시키기 위한 데이터이기 때문에 '**훈련세트**' 라고 부릅니다.

카테고리는 강아지의 종 / 색깔 / 크기 등 기본적인 것부터 시작해서 표정 / 자세 / 개체수 / 사람 유무 / 얼굴 유무 등 점차 세부적으로 넓혀 나갔고, **하나의 카테고리당 최소 2 천 장의 데이터를 수집**했습니다.





선정 카테고리	카테고리 별 클래스		
ZAA. 반려동물 수	없다		
	개 1마리 있다		
	고양이 1마리 있다		
	개+고양이 2마리 이상		
ZAB. 사람 유무	사람이 없다		
ZAC. 사람 얼굴 유무	사람이 있다		
ZAC. 사람 얼굴 유무	사람 얼굴이 없다		
ZAC. 사람 얼굴 유무	사람 얼굴이 있다		
ZBA. 실내/야외 여부	실내(자 포함)		
ZBA. 실내/야외 여부	야외(하늘, 풀숲, 피크닉 등)		
ABA. 얼굴 유무	없다(보이지 않는다)		
ABA. 얼굴 유무	있다		
ABH. 표정	기타/다른 표정		
	무표정		
	웃고 있는		
	놀란		
	눈치보는		
	억울한/불쌍한		
	하품하는		
	자는		
ACA. 몸통 유무	화난		
	슬픈, 아련		
	없다(보이지 않는다)		
	있다		
ACE. 정적 자세	기타/다른 자세		
	일어서 있다		
	두 뒷다리만 앉아있다		
	앞/뒷다리 둘 다 앉아있다(옆어져있다)		
	누워 있다(대체로 옆으로)		
	배를 발라당 까고 있다		
	두 발로 서 있다		

그림3. 훈련세트 수집을 위한 카테고리 분류 및 샘플데이터

데이터를 수집할 때는 단순히 검색을 통해 수집하기도 하지만, 수집을 맡은 팀원들이 데이터의 목적, 즉 앱의 방향성에 대해 앱 기획자와 충분한 논의를 한 후에 진행하는 것이 좋습니다. 우리는 실제로 사용자의 스마트폰에 있는 반려견의 사진을 분류해내야 했기 때문에, **‘일반 사용자들이 직접 찍었을 법한 이미지’**라는 기준을 두고 데이터 수집을 진행했습니다. 현재는 크롤러를 활용한 이미지 수집도 병행하고 있지만, 초기에는 카테고리와 앱 기획의도에 맞는 데이터들을 사람이 직접 판단하여 수집하고, 축적하는 과정이 필요했습니다.

(2) 데이터 멀티태깅 (Multi-tagging of Data)

수집해서 쌓아놓은 데이터를 그대로 활용하기는 어렵습니다. 데이터를 수집했다면, 이 데이터가 무엇을 의미하는지도 함께 알려줘야 AI가 스스로 익히고 훈련할 수 있겠죠. 예를 들어 이 강아지의 종은 진돗개고, 앉아있고, 얼굴은 보이고, 등 다양한 태그 또는 라벨을 붙일 수 있습니다. 하나의 데이터에

여러 개의 태그를 붙이는 작업을 우리는 ‘멀티태깅(Multi-tagging)’이라고 부릅니다.

단순해 보이지만 대량의 데이터를 적게는 수십 개, 많게는 수백 개의 태깅 기준에 따라 정확히 멀티태깅하는 작업의 난이도는 상당히 높아 그만큼 인력과 시간이 많이 필요합니다. 이러한 멀티태깅 작업을 보다 효율적으로 할 수 있도록 우리는 직접 어노테이션 툴(태깅툴)을 개발하게 됩니다. 클라우드소싱 기반의 데이터 가공 플랫폼에서 볼 수 있는 데이터 가공툴을 우리의 데이터 태깅 기준과 목적에 맞게, 빠르고 편리하게 작업할 수 있도록 개발한 것입니다.

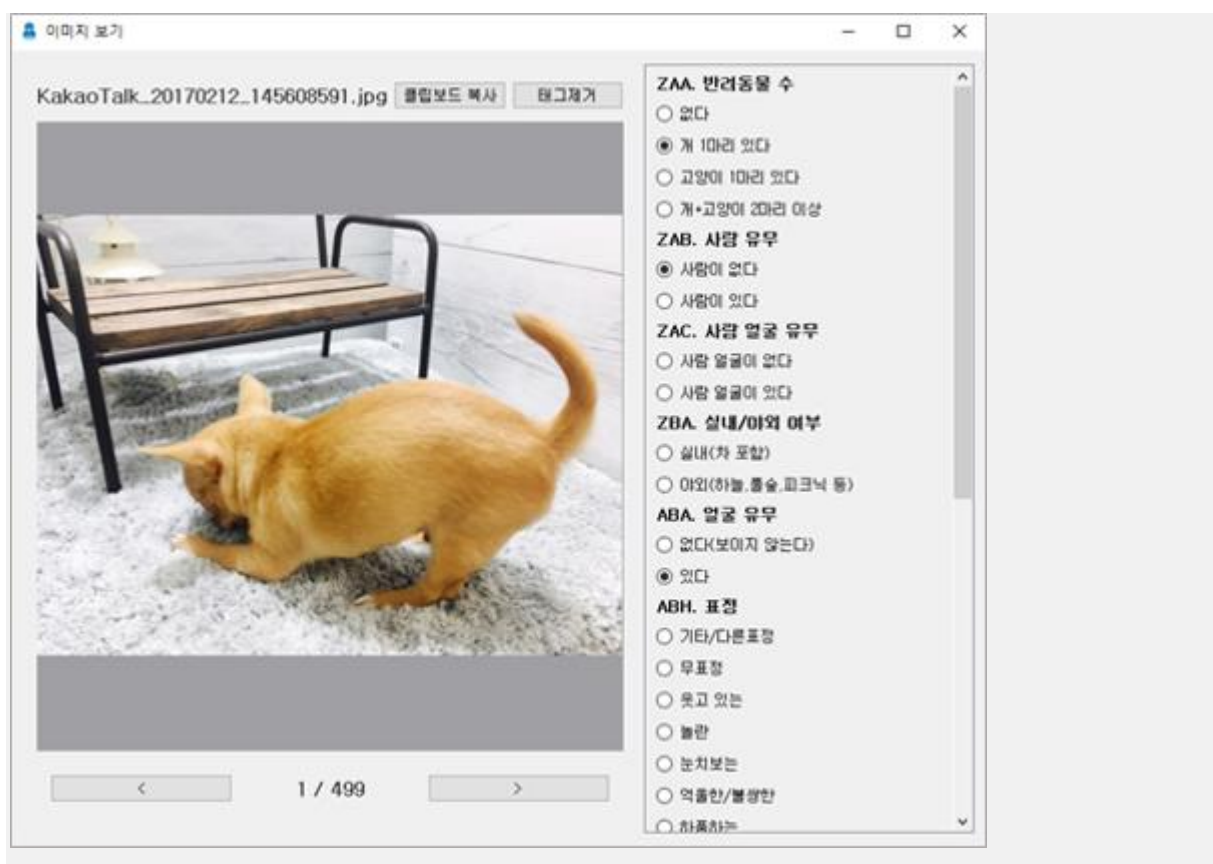


그림4. 데이터 멀티태깅을 위한 어노테이션 툴

어노테이션 툴을 통해 멀티태깅이 완료된 이미지 데이터들은 필요에 따라 한 차례에서 수차례까지 검수와 후처리를 거치게 됩니다. 수집에서 멀티태깅까지의 과정이 모두 사람이 직접 하는 작업이기 때문에 혹시 실수로 잘못 태깅하거나 빠뜨린 것은 없는지, 또는 편향된 판단에 의해 태깅하지는 않았는지 등을 확인하며 데이터를 보완하는 검수 과정입니다. 흔히 남이 범한 오류는 눈에 더 잘 띄기 때문에, 서로의 데이터를 크로스체크하면서 전체적인 데이터의 품질을

높이는 것이죠. 검수까지 마친 양질의 훈련세트는 DB 파일로 만들어져 AI 학습에 적용할 수 있게 됩니다.

(3) 평가 데이터셋과 데이터 보완(Training Data set and Complementation)

멀티태깅까지 마친 데이터를 활용해 AI 분류모델을 만들었다면 이제 '평가세트' 데이터를 통해 분류모델의 퍼포먼스를 '평가'해봐야 합니다. 다양한 데이터를 최대한 많이 수집했던 훈련세트와 달리, 평가세트는 보다 명확한 기준을 가지고 데이터를 수집하여 과연 이런 사진도 분류 가능할까? 라는 질문을 AI 모델에게 지속적으로 던지는 과정입니다.

초기 평가에서는 분류모델의 판단 오류에 의한 다양한 오분류 데이터들이 나오게 됩니다. 예를 들어, 강아지만큼 귀여운 우리 아기 사진, 인형 사진, 또는 다른 동물을 강아지 사진으로 잘못 분류하는 경우들이 발생했습니다.

오분류 데이터를 줄이기 위해서는 '강아지가 아니다' 데이터를 AI에게 학습시켜주어야 합니다. 강아지가 아닌 이미지들도 최대한 많이, 다양하게 수집하여 '이런 건 강아지가 아니다' 혹은 '이 사진엔 강아지가 없다' 라고 태깅한 후 학습시키는 것입니다. 초기에는 강아지 데이터만 대량 수집하는 것만으로도 많은 리소스가 사용되었기 때문에 강아지가 아닌 데이터들은 상대적으로 적었습니다. 그 때문에 강아지가 아닌 데이터도 강아지로 분류해 결과값이 나오는 오분류 경우의 수가 많았는데, 이러한 오분류 확률을 줄이기 위해 강아지가 아닌 이미지 데이터도 충분히 수집하는 또 한번의 데이터 수집 과정을 거치게 됩니다.



그림5. 오분류 데이터 예시 (왼쪽부터 인형, 아기, 어두운 사진)

또는 강아지가 맞지만, 종이나 표정 등 세부적인 특징을 잘 분류해내지 못한 경우에는, 그 실패 확률을 분석해서 오분류 확률이 높은 특징 카테고리의 데이터를 추가 수집, 태깅하여 재학습시키는 과정을 거칩니다. 부족한 데이터를 채워주는 데이터 보완의 과정이라고 할 수 있습니다.

분류모델의 정확도를 높이기 위한 수단으로, '임계치 조건'도 적용하기로 했습니다. 예를 들어서 강아지로 오인할 법한 사진들을 분류모델에 집어넣었더니, 강아지일 확률이 최대 0.5 가 나왔다고 가정해보겠습니다. 이때 우리는 임계치 조건을 0.8 로 설정해 두고, 확률값이 0.5 가 나왔더라도 0.8 에 미치지 못하기 때문에 강아지가 아닌 것으로 결론 짓고 포에버(PAWEVER) 앱에서 강아지로 분류하지 않도록 처리합니다. 쉽게 말해 강아지일 확률이 최소 80%는 되어야 강아지 사진으로 뽑아내도록 하는 것입니다.

마치며

지금까지 설명한 **훈련세트 데이터 수집, 데이터 멀티태깅, 평가세트 구축과 데이터 보완의 과정**을 아래 도식과 같이 정리해볼 수 있습니다. 여타 AI 데이터 구축 과정들도 많은 양의 데이터를 수집해서 가공한 후 학습을 거치는 유사한 프로세스로 진행될 것입니다. 이 과정 속에서 AI 연구자와 앱 기획자, 데이터

구축 인력들이 머리를 맞대고, 우리 앱의 퍼포먼스를 높이기 위한 최적의 데이터 구축 과정을 고민하여 커스터마이징하는 과정을 거치게 되죠.

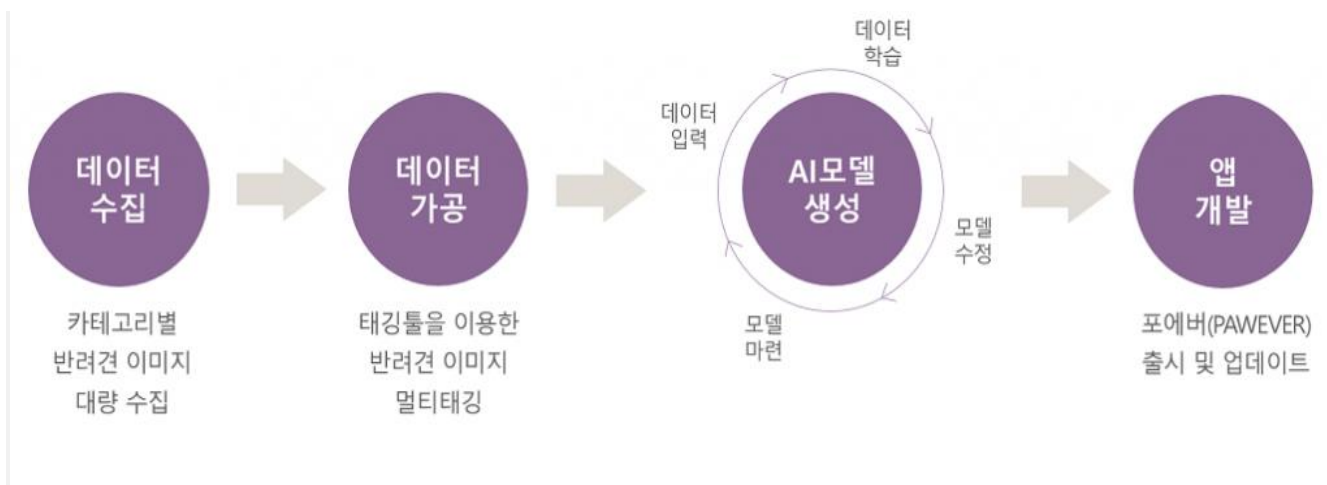


그림6. AI데이터 구축과 활용 과정 (NIA 『2019년 AI 학습용 데이터 구축』 사업 방향 및 중점 사항 자료 참고)

포에버(PAWEVER) 프로젝트에는 동일 기간에 **평균 15명** 정도의 데이터 구축 인력이 작업에 참여했습니다. 여러명이 협업하는 과정에서 기존에는 기획자 혼자 고민했던, 크고 작은 아이디어와 인사이트가 계속해서 창출되었습니다. AI의 성능을 높여주는 AI 데이터 본연의 목적 달성과 더불어, 데이터 구축 과정에서 나오는 다양한 피드백은 기획자의 기획을 보다 촘촘하게 만들어주었습니다.

포에버(PAWEVER)는 이미 출시를 마치고 애견인들에게는 빠른 속도로 인기를 얻고 있으며, 앞으로 더 재미있는 프로젝트를 준비하고 있습니다. 그 뒤에서는 여전히 분류모델의 정확도 향상과 차기모델을 준비하며 더 많은 양질의 AI 데이터를 쌓아가고 있습니다. 포에버(PAWEVER)는 물론 앞으로 진행될 다양한 AI 데이터 프로젝트들도 관심 있게 지켜봐 주시기 바랍니다. 감사합니다.

머신러닝을 위한 데이터 사전 처리: 옵션 및 권장사항

2부로 구성된 이 문서는 머신러닝(ML)을 위한 데이터 엔지니어링과 특성 추출이라는 주제를 살펴봅니다. 1부에 해당하는 이 문서에서는 Google Cloud의 머신러닝 파이프라인에서 데이터를 사전 처리하기 위한 권장사항을 설명합니다. 이 문서는 TensorFlow 및 오픈소스 TensorFlow Transform (`tf.Transform`) 라이브러리를 사용하여 데이터를 준비하고, 모델을 학습시키고, 예측 모델을

제공하는 데 초점을 맞추고 있습니다. 1 부에서는 머신러닝을 위해 데이터를 사전 처리할 때의 어려움을 집중적으로 다루고, **Google Cloud** 에서 데이터 변환을 효율적으로 수행하기 위한 옵션과 시나리오를 설명합니다.

이 문서는 사용자가 [BigQuery](#), [Dataflow](#), [AI Platform \(ML\) Engine](#), TensorFlow [Estimator](#) API 에 익숙하다고 가정합니다.

2 부에서는 `tf.Transform` 파이프라인을 구현하는 방법의 단계별 가이드를 제공합니다. 자세한 내용은 [tf.Transform 을 사용하여 Google Cloud 에서 머신러닝을 위한 데이터 사전 처리 - 2 부](#)를 참조하세요.

소개

특성을 따라잡는 것은 어렵고 시간이 많이 소요되는 작업이며, 전문가 지식이 필요합니다. '응용 머신러닝'은 기본적으로 특성 추출입니다.

—앤드류 응, 스탠퍼드 대학([출처](#))

머신러닝(ML)을 이용하면 복잡하지만 유용할 수 있는 데이터 패턴을 자동으로 찾을 수 있습니다. 이러한 패턴은 새로운 데이터 포인트에서 사용될 수 있는 ML 모델로 압축됩니다. 이러한 프로세스를 *예측하기* 또는 *추론하기*라고 합니다.

ML 모델을 구축하는 작업은 여러 단계로 이루어진 프로세스입니다. 단계마다 저마다의 기술적 및 개념적 문제가 있습니다. 2 부로 구성된 이 문서에서는 소스 데이터를 선택, 변환, 강화하여 대상(응답) 변수(감독되는 학습 작업에 있음)에 대한 강력한 예측 신호를 만드는 프로세스를 중점적으로 다룹니다. 이 작업에는 도메인 지식과 데이터 사이언스 기술이 결합되며, [특성 추출](#)의 핵심입니다.

실제 ML 모델의 학습 데이터세트 크기는 테라바이트(TB) 단위에 쉽게 도달하거나 이를 초과할 수 있습니다. 따라서 이러한 데이터세트를 효율적이고 분산적인 방식으로 처리하려면 대규모 데이터 처리 프레임워크가 필요합니다. 또한, 예측을 위해 ML 모델을 사용할 때는 라이브 데이터세트가 모델이 예상하는 방식대로 ML 모델에 제공되도록 학습 데이터에 사용한 것과 동일한 변환을 새로운 데이터 포인트에 적용해야 합니다.

첫 번째 문서에서는 특성 추출 작업의 다양한 세부사항 수준(인스턴스 수준, 전체 전달, 기간 집계)에서 발생하는 문제에 대해 설명합니다. 또한 이 문서는 **Google Cloud** 에서 ML 을 위한 데이터 변환을 효율적으로 수행하기 위한 옵션과 시나리오를 설명합니다.

이 문서에서는 [TensorFlow Transform](#)(`tf.Transform`), 즉 데이터 사전 처리 파이프라인을 통해 인스턴스 수준 및 전체 전달 데이터 변환을 모두 정의할 수 있는 TensorFlow 용 라이브러리의 개요도 제공합니다. 이러한

파이프라인은 [Apache Beam](#)으로 실행되며 아티팩트가 부산물로 생성되어 모델이 제공될 때와 동일한 예측 도중 변환을 적용합니다.

머신러닝을 위한 데이터 사전 처리

이 섹션에서는 데이터 사전 처리 작업과 데이터 준비 단계를 소개합니다. 사전 처리 작업의 유형과 세부사항에 대해서도 설명합니다.

데이터 엔지니어링과 특성 추출 비교

ML을 위한 데이터 사전 처리에는 데이터 엔지니어링과 특성 추출이 모두 포함됩니다. 데이터 엔지니어링은 *원시 데이터를 준비된 데이터*로 변환하는 프로세스입니다. 그다음에는 특성 추출을 통해 준비된 데이터를 미세 조정하여 ML 모델에서 예상하는 특성을 만듭니다. 이 용어들의 자세한 의미는 다음 목록에 설명되어 있습니다.

원시 데이터(또는 간단히 데이터)

ML을 위한 사전 준비가 전혀 되지 않은 소스 형식의 데이터를 의미합니다. 이러한 맥락에서 데이터는 원시 형태(데이터 레이크에 있음)이거나 변환된 형태(데이터 웨어하우스에 있음)일 수 있습니다. 데이터 웨어하우스에 있는 변환된 데이터는 분석에 사용되기 위해 원래의 원시 형태에서 변환되었을 수도 있습니다. 하지만 이 맥락에서는 데이터가 ML 작업용으로 특별히 준비되지 않았음을 의미합니다. 또한 *최종적으로* 예측을 위해 ML 모델을 호출하는 스트리밍 시스템에서 보낸 데이터도 원시 형태의 데이터로 간주됩니다.

준비된 데이터

ML 작업용으로 준비된 데이터세트 형식을 의미합니다. 데이터 소스는 파싱되고, 결합되고, 테이블 형태로 제작되었습니다. 데이터가 적절한 수준으로 집계되고 요약되었습니다. 예를 들어 데이터세트의 각 행은 고유한 고객을 나타내고, 각 열은 고객의 요약 정보(예: 지난 6주 동안 지출한 총액)를 나타냅니다. 감독되는 학습 작업의 경우 대상 특성이 존재합니다. 관련 없는 열은 삭제되었으며 유효하지 않은 레코드는 필터링되었습니다.

추출된 특성

모델에서 예상하는 미세 조정된 특성이 있는 데이터셋을 나타냅니다. 즉, 준비된 데이터셋의 열에서 특정 ML 전용 작업을 수행하고, 뒷부분의 [사전 처리 작업](#)에 설명된 대로 학습 및 예측 중에 모델의 새로운 특성을 만듭니다. 숫자 열을 0 과 1 사이의 값으로 조정하고, 값을 자르고, 카테고리 특성을 원-핫 인코딩하는 것을 예로 들 수 있습니다.

그림 1 은 관련된 단계를 보여줍니다.

그림 1. 원시 데이터에서 준비된 데이터, 추출된 특성, 머신러닝으로의 데이터 흐름

실제로 소스가 같은 데이터의 준비 단계는 각기 다른 경우가 종종 있습니다. 예를 들어 데이터 웨어하우스에 있는 테이블의 한 필드를 추출된 특성으로 바로 사용할 수 있습니다. 한편, 같은 테이블의 다른 필드는 변환을 거쳐야 추출된 특성이 될 수 있습니다. 마찬가지로 데이터 엔지니어링 및 특성 추출 작업이 같은 데이터 사전 처리 단계에 결합될 수도 있습니다.

사전 처리 작업

데이터 사전 처리에는 다양한 작업이 포함됩니다. 각 작업의 목표는 머신러닝이 더 나은 예측 모델을 만들 수 있도록 돕는 것입니다. 이러한 사전 처리 작업의 세부정보는 이 문서의 범위를 벗어나는 것이지만, 참조를 위해 몇몇 작업은 이 섹션에 간단하게 설명되어 있습니다.

구조화된 데이터의 경우, 데이터 사전 처리 작업에는 다음이 포함됩니다.

- **데이터 정리.** 손상되거나 유효하지 않은 값이 있는 레코드를 삭제하거나 원시 데이터로 수정하는 프로세스 그리고 다수의 열이 누락된 레코드를 삭제하는 프로세스입니다.
- **인스턴스 선택 및 파티션 나누기.** 입력 데이터셋에서 데이터 포인트를 선택하여 [학습](#), [평가\(검증\)](#), [테스트 세트](#)를 만드는 프로세스입니다. 이 프로세스는 반복 가능한 무작위 샘플링, 소수 클래스 오버샘플링, 계층화된 파티션 나누기를 위한 기술을 포함하고 있습니다.
- **특성 미세 조정.** 숫자 값 조정 및 정규화, 누락 값 입력, 이상점 자르기, 비대칭 분포로 값 조정 등 ML 을 위한 특성의 품질을 개선합니다.
- **표현 변환.** 버킷화를 통해 숫자 특성을 카테고리 특성으로 변환하고 [원-핫 인코딩](#), [카운트를 통한 학습](#), 희소 특성 임베딩 등을 통해 카테고리 특성을 숫자 표현으로 변환합니다. 일부 모델에서는 숫자 및 카테고리 특성 중에 하나만 사용할 수 있으며, 다른 일부는 혼합 유형 특성을 처리할 수 있습니다. 모델이 두

유형을 모두 처리하는 경우에도 같은 특성의 다른 표현(숫자 및 카테고리)을 활용할 수 있습니다.

- **특성 추출.** [PCA](#), [임베딩](#) 추출, [해싱](#) 등의 기술을 사용하여 더 작은 크기의 더 강력한 데이터 표현을 만들어서 특성 수를 줄이는 프로세스입니다.
- **특성 선택.** [필터 또는 래퍼 메소드](#)를 사용하여 모델을 학습시키기 위한 입력 특성 하위 집합을 선택하고 관련이 없거나 중복된 특성을 무시하는 프로세스입니다. 특성에 많은 값이 누락된 경우 특성을 간단히 삭제하는 것도 포함될 수도 있습니다.
- **특성 생성.** 일변량 수학 함수를 사용한 [다항식 전개](#) 또는 특성 상호작용을 포착하기 위한 [특성 교차](#) 같은 일반적인 기법을 사용하여 새로운 특성을 만듭니다. ML 사용 사례 도메인의 비즈니스 논리를 사용하여 특성을 생성할 수도 있습니다.

구조화되지 않은 데이터(예: 이미지, 오디오, 텍스트 문서)를 작업할 때는 도메인 지식 기반 특성 추출이 모델 아키텍처에 통합되기 때문에 딥 러닝이 이러한 특성 추출을 수행하지 않아도 됩니다. [컨볼루션 레이어](#)는 자동 특성 사전 처리기입니다. 그렇기 때문에 올바른 모델 아키텍처를 생성하려면 데이터에 대한 경험적 지식이 필요합니다. 또한 다음과 같은 몇 가지 사전 처리가 필요합니다.

- 텍스트 문서의 경우 [어간 추출 및 원형 추출](#), [TF-IDF](#) 계산, [n-gram](#) 추출, 임베딩 조화
- 이미지의 경우 클리핑, 크기 조절, 자르기, 가우스 블러, 카나리아 필터
- 학습을 모두 마친 모델의 마지막 레이어를 제외한 모든 것을 특성 추출 단계로 취급하는 [전이 학습](#). 텍스트와 이미지를 포함한 모든 유형의 데이터에 적용됩니다.

사전 처리 세부사항

이 섹션에서는 데이터 변환 유형의 세부사항에 대해 설명합니다. 학습 데이터에 적용되는 변환을 사용하여 새로운 예측용 데이터 포인트를 준비할 때 이 관점이 중요한 이유를 보여줍니다.

사전 처리 및 변환 작업은 작업 세부사항에 따라 다음과 같이 분류될 수 있습니다.

- **학습 및 예측 도중의 인스턴스 수준 변환.** 같은 인스턴스(데이터 포인트)의 값만 변환에 필요한 간단한 변환입니다. 특성 값을 특정 기준에 맞춰 자르거나, 다른 특성의 다항식으로 전개하거나, 두 특성을 곱하거나, 두 특성을 비교하여 부울 플래그를 만드는 것을 예로 들 수 있습니다.

원시 입력값이 아니라 변환된 특성을 모델에 학습시키는 것이기 때문에 이러한 변환은 학습 및 예측 도중에 동일하게 적용되어야 합니다. 데이터가 동일하게

변환되지 않을 경우, 모델은 학습되지 않은 값이 분포된 데이터를 받게 되기 때문에 제대로 작동하지 못합니다. 자세한 내용은 이 문서의 뒷부분에 있는 [사전 처리 과제](#)에서 학습/제공 편향에 대한 설명을 참조하세요.

- **학습 도중에는 전체 전달 변환, 하지만 예측 도중에는 인스턴스 수준 변환.** 이 시나리오에서는 상태 저장 변환이 이루어지지만 몇 가지 미리 계산된 통계를 사용하여 변환해야 합니다. 학습 도중에는 학습 데이터 전체를 분석하여 예측 시 학습 데이터, 평가 데이터, 새 데이터를 변환하기 위한 수량(최솟값, 최댓값, 평균, 편차 등)을 계산해야 합니다.

예를 들어 학습을 위해 숫자 특성을 정규화하려면 학습 데이터 전체의 평균(μ) 및 표준편차(σ)를 계산해야 합니다. 이것을 *전체 전달*(또는 *분석*) 작업이라고 합니다. 그런 다음 예측을 위한 모델을 제공할 때, 새 데이터 포인트의 값을 정규화하여 학습/제공 편향을 방지해야 합니다. 따라서 학습 도중에 계산되는 μ 및 σ 값은 특성 값을 조정하는 데 사용됩니다. 이 작업은 간단한 *인스턴스 수준* 작업입니다.

$$\text{valuescaled} = (\text{valueraw} - \mu) \div \sigma$$

이 카테고리에 속하는 변환은 다음과 같습니다.

- 학습 데이터세트에서 계산된 *min* 및 *max*를 사용하는 숫자 특성의 **MinMax** 조정
- 학습 데이터세트에서 계산된 μ 및 σ 를 사용하는 숫자 특성의 표준 조정(**z-점수** 정규화)
- 백분위 수를 사용하는 숫자 특성의 버킷화
- 중앙값(숫자 특성) 또는 모드(카테고리 특성)를 사용하는 누락된 값 입력
- 입력 카테고리 특성의 모든 개별 값(어휘)을 추출하여 문자열(명목값)에서 정수(색인)로 변환
- 모든 문서(인스턴스)에서 용어(특성 값) 등장 횟수를 집계해 **TF-IDF**를 계산
- 데이터를 하위 차원의 공간에 투영하기 위해 입력 특성의 **PCA** 계산(선형 종속 특성 사용)

학습 데이터만 사용하여 μ , σ , *min*, *max* 등의 통계를 계산해야 합니다. 이러한 작업에 테스트 및 평가 데이터를 추가하는 경우에는 평가 및 테스트 데이터의 [정보를 유출](#)해 모델을 학습시키게 됩니다. 이러한 행위는 테스트 및 평가 결과의 신뢰도에 영향을 줍니다. 또한 모든 데이터세트에 일관된 변환이 적용되도록 학습 데이터에서 계산된 동일한 통계를 사용하여 테스트 및 평가 데이터를 변환합니다.

- **학습 및 예측 도중의 기간 집계.** 이 접근 방식을 사용하려면 시간 경과에 따른 실시간 값을 요약하여 특성을 만들어야 합니다. 즉, 집계할 인스턴스가 임시 **window** 절을 통해 정의됩니다. 예를 들어 지난 5분, 지난 10분, 지난 30분, 기타 간격 동안의 경로 교통정보 측정항목을 기준으로 택시 이동 시간을 예측하는 모델을 학습시키고자 한다고 가정해보세요. 지난 3분 동안 계산된 온도

및 진동 값의 이동 평균을 기준으로 엔진 부품의 장애를 예측하는 작업을 또 다른 예로 들 수 있습니다. 이러한 집계는 학습을 위해 오프라인에서 준비될 수는 있지만, 제공하는 동안 데이터 스트림에서 실시간으로 계산되어야 합니다.

더 정확히 말하면 학습 데이터를 준비할 때 원시 데이터 안에 집계된 값이 없으면 데이터 엔지니어링 단계 중에 값이 생성됩니다. 원시 데이터는 주로 `(entity, timestamp, value)` 형식의 데이터베이스에 저장됩니다. 앞의 예시에서 `entity`는 택시 경로의 경우 경로 구간 식별자이고, 엔진 장애의 경우 엔진 부품 식별자입니다. 기간 설정 작업을 사용하여 `(entity, time_index, aggregated_value_over_time_window)`를 계산하고 집계 특성을 모델 학습을 위한 입력으로 사용할 수 있습니다.

하지만 실시간(온라인) 예측 모델이 제공되고 있을 때는 모델이 집계된 값에서 추출된 특성을 입력으로 예상합니다. 따라서 **Apache Beam**과 같은 스트림 처리 기술을 사용하여 시스템으로 스트리밍되는 실시간 데이터 포인트에서 집계를 바로 계산할 수 있습니다. 학습 및 예측 전에 이러한 집계에 추가적인 특성 추출(미세 조정)을 수행할 수도 있습니다.

Google Cloud의 머신러닝 파이프라인

이 섹션에서는 관리형 서비스를 사용하여 **Google Cloud**에 **TensorFlow ML** 모델을 학습시키고 제공하는 일반적인 엔드 투 엔드 파이프라인의 구성요소를 설명합니다. 데이터 사전 처리 작업의 여러 카테고리를 구현할 수 있는 위치와 이러한 변환을 구현할 때 발생할 수 있는 일반적인 과제에 대해서도 설명합니다. 뒷부분에서는 **TensorFlow Transform** 라이브러리가 이러한 문제를 해결하는 데 어떤 도움을 주는지 확인할 수 있습니다.

고수준 아키텍처

그림 2는 **TensorFlow** 모델을 학습시키고 제공하기 위한 일반적인 ML 파이프라인의 고수준 아키텍처를 보여줍니다. 다이어그램에 있는 라벨 **A, B, C**는 파이프라인에서 데이터 사전 처리가 수행될 수 있는 여러 위치를 나타냅니다. 이 단계에 대한 세부정보는 뒤에 나오는 섹션에 설명되어 있습니다.

그림 2. Google Cloud에서의 ML 학습 및 제공을 위한 고수준 아키텍처

이 파이프라인은 다음과 같은 단계로 구성됩니다.

1. 가져온 원시 데이터는 **BigQuery**(이미지, 문서, 오디오, 비디오 등의 경우에는 **Cloud Storage**)에 저장됩니다.

2. 데이터 엔지니어링(준비) 및 특성 추출은 **Dataflow** 를 사용하여 규모에 맞춰 실행됩니다. 이를 통해 **ML** 에서 사용 가능한 학습, 평가, 테스트 세트가 생성되고 **Cloud Storage** 에 저장됩니다. 이상적으로 이러한 데이터세트는 **TensorFlow** 계산에 최적화된 형식인 파일로 저장됩니다.
3. **TensorFlow** 모델 [트레이너 패키지](#)는 **AI Platform** 에 제출되고, **AI Platform** 은 이전 단계의 사전 처리된 데이터를 사용하여 모델을 학습시킵니다. 이 단계의 출력은 **Cloud Storage** 로 내보내는 학습된 **TensorFlow** [저장된 모델](#)입니다.
4. 학습된 **TensorFlow** 모델은 온라인 예측에 사용되도록 **REST API** 가 있는 마이크로서비스로 **AI Platform** 에 배포됩니다. 동일한 모델을 일괄 예측 작업에 사용할 수도 있습니다.
5. 모델이 **REST API** 로 배포된 후, 클라이언트 앱 및 내부 시스템은 일부 데이터 포인트가 있는 요청을 보내고 모델로부터 예측이 포함된 응답을 받아 이 **API** 를 호출할 수 있습니다.
6. 이 파이프라인을 조정하고 자동화하기 위해 [Cloud Composer](#) 를 스케줄러로 사용하여 데이터 준비, 모델 학습, 모델 배포 단계를 호출할 수 있습니다.

사전 처리를 수행할 위치

그림 2에 나타난 라벨 A, B, C와 같이 데이터 사전 처리 작업은 **BigQuery**, **Dataflow**, **TensorFlow** 의 세 곳에서 수행될 수 있습니다.

옵션 A: BigQuery

일반적으로 다음 작업의 논리는 **BigQuery** 에서 구현됩니다.

- 샘플링: 데이터에서 하위 집합을 임의로 선택합니다.
- 필터링: 관련이 없거나 유효하지 않은 인스턴스를 삭제합니다.
- 파티션 나누기: 데이터를 분할하여 학습, 평가, 테스트 세트를 만듭니다.

BigQuery SQL 스크립트는 **Dataflow** 사전 처리 파이프라인의 소스 쿼리로 사용될 수 있습니다. 이 작업이 그림 2의 데이터 사전 처리 단계입니다. 예를 들어 시스템은 캐나다에서 사용될 예정이지만 데이터 웨어하우스에는 전 세계의 트랜잭션이 포함되어 있다고 가정합니다. 캐나다 전용 학습 데이터를 필터링하는 것이 **BigQuery** 에서 가장 효과적입니다.

BigQuery SQL 스크립트에서 인스턴스 수준 변환, 상태 저장 전체 전달 전환, 기간 집계 특성 변환을 구현하여 학습 데이터를 준비하는 것이 가능합니다. 그러나 이 방법은 권장하지 않습니다.

온라인 예측을 위한 모델을 배포하는 경우에는 다른 시스템에서 생성했으며 모델의 **API**에 배포될 원시 데이터 포인트에서 **SQL** 사전 처리 작업을 복제해야 합니다. 다시 말해, 논리를 두 번 구현해야 합니다. 첫 번째는 **BigQuery**에서 학습 데이터를 사전 처리하기 위해 **SQL**에서 구현합니다. 두 번째는 모델을 사용하여 예측을 위해 온라인 데이터 포인트를 사전 처리하는 앱의 논리에서 구현합니다. 예를 들어 클라이언트 앱이 자바로 작성된 경우에는 자바에서 논리를 다시 구현해야 합니다. 이로 인해 구현 불일치로 인한 오류가 발생할 수 있습니다. 이 문서의 뒷부분에 있는 [사전 처리 과제](#) 섹션에서 학습/제공 편향에 대한 설명을 참조하세요.

또한 두 가지 구현을 유지하려면 추가적인 부담이 생깁니다. 학습 데이터를 사전 처리하기 위해 **SQL**에서 논리를 변경할 때마다 제공 시점의 데이터를 사전 처리하는 데 알맞게 자바 구현을 변경해야 합니다.

[AI Platform 일괄 예측](#)을 통해 모델을 일괄 예측(스코어링)에만 사용하며 스코어링용 데이터의 소스가 **BigQuery**인 경우, 이러한 사전 처리 작업을 **BigQuery SQL** 스크립트의 일부로 구현하는 방식도 가능합니다. 이렇게 하면 동일한 사전 처리 **SQL** 스크립트를 사용하여 학습 데이터와 스코어링 데이터를 모두 준비할 수 있습니다. 즉, 숫자 특성을 조정하기 위한 평균이나 편차 등 상태 저장 변환에 필요한 수량을 보조 테이블을 사용하여 저장하고 있습니다. 또한 **SQL** 스크립트가 좀 더 복잡해지고, 학습 및 점수 **SQL** 스크립트가 복잡하게 종속됩니다.

다음 코드 스니펫은 학습 및 예측을 위해 데이터를 준비하는 **BigQuery SQL**의 예시를 가정해서 보여줍니다. 첫 번째 스크립트(학습 스크립트)에서 필드 **f1** 및 **f2**의 평균과 표준편차는 **stats** 테이블에 저장됩니다. 그런 다음 **training_data** 테이블을 **stats** 테이블과 결합하여 학습 데이터를 변환합니다.

```
# computing stats for normalizing numerical features: f1 and f2
CREATE OR REPLACE TABLE stats AS
(
  SELECT
    AVG(f1) AS f1_mean,
    STDDEV(f1) AS f1_stdv,
    AVG(f2) AS f2_mean,
    STDDEV(f2) AS f2_stdv
  FROM
    my_dataset.training_data
)
# extracting data for training
SELECT
  (data.f1 - stats.f1_mean)/stats.f1_stdv AS f1_NORMALIZED,
  (data.f2 - stats.f2_mean)/stats.f2_stdv AS f2_NORMALIZED,
  POWER((data.f1 - stats.f1_mean)/stats.f1_stdv, 2) AS
```

```

f1_NORMALIZED_SQUARED,
    POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS
f2_NORMALIZED_SQUARED,
    ((data.f1 - stats.f1_mean)/stats.f1_stdv) * ((data.f2 -
stats.f2_mean)/stats.f2_stdv) AS f1_X_f2
FROM
    my_dataset.training_data AS data
CROSS JOIN
    stats
WHERE #filtering data for training
    data.entry_year = 2018
AND
    data.entry_location = 'Canada'

```

다음 목록은 예측 스크립트를 보여줍니다. 이 스크립트에서는 stats 테이블이 prediction_data 테이블과 결합되어 점수를 위한 데이터를 준비합니다. 마찬가지로 다른 인스턴스 수준 변환은 복제됩니다.

```

#extracting data for prediction
SELECT
    (data.f1 - stats.f1_mean)/stats.f1_stdv AS f1_NORMALIZED,
    (data.f2 - stats.f2_mean)/stats.f2_stdv AS f2_NORMALIZED,
    POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS
f1_NORMALIZED_SQUARED,
    POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS
f2_NORMALIZED_SQUARED,
    ((data.f1 - stats.f1_mean)/stats.f1_stdv) * ((data.f2 -
stats.f2_mean)/stats.f2_stdv) AS f1_X_f2
FROM
    my_dataset.prediction_data AS data
CROSS JOIN
    stats

```

옵션 B: Dataflow

그림 2에 나와 있듯이 계산 비용이 높은 사전 처리 작업을 Apache Beam에서 구현하고, 일괄 및 스트림 데이터 처리를 위한 완전 관리형 자동 확장 서비스인 Dataflow를 사용하여 규모에 맞춰 실행할 수 있습니다.

Dataflow는 인스턴스 수준 변환, 스테이트풀(Stateful) 전체 전달 변환, 기간 집계 특성 변환을 수행할 수 있습니다. 특히 ML

모델이 total_number_of_clicks_last_90sec와 같은 입력 특성을 예상하는 경우 Apache Beam [윈도우 함수](#)는 클릭수와 같은 실시간(스트리밍) 이벤트

데이터의 기간 값을 집계하여 해당 특성을 계산할 수 있습니다. 앞서 설명한 [변환 세부사항](#)에서는 이를 '학습 및 제공 도중 기간 집계'라고 지칭했습니다.

그림 3은 실시간에 가까운 예측을 위해 스트림 데이터를 처리할 때의 **Dataflow** 역할을 보여줍니다. 기본적으로 이벤트(데이터 포인트)는 [Pub/Sub](#)에 수집됩니다. **Dataflow**는 이러한 데이터 포인트를 사용하고, 시간 경과에 따른 집계를 바탕으로 특성을 계산하고, 예측을 위해 배포된 **ML** 모델 **API**를 호출합니다. 그런 다음 예측을 발신 **Pub/Sub** 큐에 보냅니다. 여기에서 예측은 다운스트림(모니터링 또는 제어) 시스템에 의해 사용될 수도 있고, 알림 등을 통해 원래의 요청 클라이언트로 전달될 수도 있습니다. 실시간 가져오기를 위해 [Cloud Bigtable](#)과 같이 지연 시간이 짧은 데이터 저장소에 예측을 저장할 수도 있습니다. **Cloud Bigtable**은 예측이 필요할 때 조회할 수 있도록 이러한 실시간 집계를 축적하고 저장하는 데 사용될 수도 있습니다. 그림 3은 이 시나리오를 보여줍니다.

같은 **Apache Beam** 구현을 사용하여 **BigQuery** 같은 오프라인 **Datastore**에서 오는 학습 데이터를 일괄 처리하고, 온라인 예측을 제공하기 위한 실시간 데이터를 스트림 처리할 수 있습니다.

그림 3. Dataflow에서 스트림 데이터를 사용하여 예측하는 고수준 아키텍처

다른 일반적인 아키텍처에서는 클라이언트 앱이 온라인 예측을 위해 배포된 모델 **API**를 직접 호출합니다(앞의 그림 2 참조). 그럴 경우에는 학습 데이터를 준비하기 위해 **Dataflow**에 구현된 사전 처리 작업이 모델로 직접 이동하는 예측 데이터에 적용되지 않습니다. 따라서 이러한 변환은 온라인 예측을 제공하는 동안 모델의 핵심 부분이어야 합니다.

옵션 C: TensorFlow

그림 2에서 볼 수 있듯이 **TensorFlow** 모델 자체에 데이터 사전 처리 및 변환 작업을 구현할 수 있습니다. 그림에 나와 있듯이 **TensorFlow** 모델을 학습시키기 위해 구현하는 사전 처리는 예측을 위해 모델을 내보내고 배포할 때 모델에 포함되는 핵심 부분이 됩니다. 다음 방법 중 하나를 통해 **TensorFlow** 변환을 수행할 수 있습니다.

- `crossed_column`, `embedding_column`, `bucketized_column` 등을 사용하여 기본 [feature columns](#)를 확장합니다.
- 세 가지 [입력 함수](#)(`train_input_fn`, `eval_input_fn`, `serving_input_fn`) 모두에서 호출하는 함수에 모든 인스턴스 수준 변환 논리를 구현합니다.
- [맞춤 에스티메이터](#)를 만드는 경우 `model_fn` 함수에 코드를 넣습니다.

온라인 예측을 위해 `SavedModel` 의 제공 인터페이스를 정의하는 `serving_input_fn` 함수에서 동일한 변환 논리 코드를 사용하는 경우, 이 함수는 학습 데이터를 준비하는 데 사용된 것과 동일한 변환이 제공 도중의 새로운 예측 데이터 포인트에 적용되도록 합니다.

하지만 앞서 강조한 바와 같이 전체 전달 변환은 **TensorFlow** 모델에서 구현할 수 없습니다.

사전 처리 과제

다음은 데이터 사전 처리를 구현할 때 마주하게 되는 주요 과제입니다.

- **학습-제공 편향.** 학습-제공 편향은 학습 도중과 제공 도중의 효율성(예측 성능) 차이를 의미합니다. 이 편향은 학습 및 제공 파이프라인에서 데이터를 처리하는 방법의 불일치로 인해 발생할 수 있습니다. 예를 들어 대수적으로 변환된 특성을 모델에 학습시켰지만 제공 도중에 원시 특성이 포함되는 경우, 예측 출력이 정확하지 않을 수 있습니다.

앞서 옵션 C: TensorFlow 에서 설명한 것처럼 모델 자체에 변환이 포함될 경우에는 인스턴스 수준 변환을 처리하는 것이 간단할 수 있습니다. 이 경우에는 모델 제공 인터페이스가 원시 데이터를 예상하는 반면, 모델은 출력을 계산하기 전에 이 데이터를 내부에서 변환합니다. 원시 학습 및 예측 데이터 포인트에 동일한 변환이 적용됩니다.

- **전체 전달 변환.** **TensorFlow** 모델에 변환을 구현하는 작업은 조정 및 정규화 같은 변환과 함께 사용될 수 없습니다. 앞에 나온 옵션 B: Dataflow 에서 설명한 것처럼, 이러한 변환에서는 일부 통계(예: 숫자 특성을 조정하기 위한 *max* 및 *min* 값)를 학습 데이터로 미리 계산해야 합니다. 그런 다음 이 값은 예측을 위한 모델 제공 중에 인스턴스 수준 변환으로 사용되어 새 원시 데이터 포인트를 변환할 위치에 저장됩니다. 이로써 학습-제공 편향이 방지됩니다.
- **더 나은 학습 효율을 위해 데이터 미리 준비.** 인스턴스 수준 변환을 모델의 일부로 구현하면 학습 프로세스의 효율성에 영향을 미칠 수 있습니다. 1,000 개의 특성을 포함하는 원시 학습 데이터가 있고 10,000 개의 특성을 생성하기 위해 인스턴스 수준 변환을 혼합해 적용한다고 가정해 보겠습니다. 이러한 변환을 모델의 일부로 구현한 후에 모델에 원시 학습 데이터를 공급하면 이러한 10,000 개의 작업이 각 인스턴스에 N 번씩 적용됩니다. 여기서 N 은 세대 수입니다. 그 외에도 액셀러레이터(**GPU** 또는 **TPU**)를 사용 중인 경우, **CPU** 가 이러한 변환을 수행하는 동안에는 액셀러레이터가 유휴 상태이기 때문에 비싼 액셀러레이터를 효율적으로 사용할 수 없습니다.

이상적으로 학습 데이터는 옵션 B: Dataflow 에서 설명한 기술을 사용하여 학습 전에 변환됩니다. 이때 10,000 개 변환 작업은 각 학습 인스턴스에 한 번씩만 적용됩니다. 변환된 학습 데이터는 이후 모델에 제공됩니다. 더 이상의 변환은 적용되지 않으며, 액셀러레이터는 항상 사용됩니다. 또한 **Dataflow** 를 사용하면

완전 관리형 서비스를 사용하여 대량의 데이터를 규모에 맞춰 사전 처리하는 데 도움이 됩니다.

학습 데이터를 미리 준비하는 것이 학습 효율성을 높이는 데 도움이 된다는 점을 확인할 수 있습니다. 그러나 변환 논리를 모델 외부에 구현([옵션 A: BigQuery](#) 또는 [옵션 B: Dataflow](#)에서 설명한 접근 방식)해도 학습-제공 편향 문제가 해결되지는 않습니다. 이 변환 논리는 예측을 위해 제공되는 새로운 데이터 포인트에 적용될 위치에 구현되어야 합니다. 이제 모델 인터페이스가 변환된 데이터를 예상하기 때문입니다. `TensorFlow Transform`(`tf.Transform`) 라이브러리가 이 문제를 해결할 수 있습니다.

tf.Transform 작동 원리

`tf.Transform` 라이브러리는 전체 전달이 필요한 변환에 유용합니다. `tf.Transform`의 출력은 학습 및 제공에 사용될 수 있도록 인스턴스 수준 변환 논리는 물론 전체 전달 변환에서 계산된 통계도 나타내는 **TensorFlow** 그래프로 보내집니다. 학습 및 제공에 같은 그래프를 사용하면 동일한 변환이 두 단계에 모두 적용되기 때문에 편향을 방지할 수 있습니다. 또한 `tf.Transform`은 **Dataflow**의 일괄 처리 파이프라인에서 대규모로 실행되어 학습 데이터를 미리 준비하고 학습 효율을 높일 수 있습니다.

다음 다이어그램은 `tf.Transform`이 학습 및 예측을 위해 데이터를 사전 처리하고 변환하는 동작을 보여줍니다.

그림 4. 데이터를 사전 처리하고 변환하는 `tf.Transform`의 동작

학습 및 평가 데이터 변환

`tf.Transform` Apache Beam API에 구현된 변환을 사용하고 **Dataflow**에서 이를 대규모로 실행하여 원시 학습 데이터를 사전 처리합니다. 사전 처리는 두 단계로 진행됩니다.

- **분석 단계.** 분석 단계 중에는 스테이트풀(Stateful) 변환의 필수 통계(예: 평균, 편차, 백분위 수 등)가 학습 데이터와 관련된 전체 전달 작업과 함께 계산됩니다. 이 단계는 `transform_fn`을 포함한 일련의 변환 아티팩트를 생성합니다. `transform_fn`은 변환 논리를 인스턴스 수준 작업으로 사용하고 분석 단계에서 계산된 통계를 상수로 포함하는 **TensorFlow** 그래프입니다.
- **변환 단계.** 변환 단계 중에는 `transform_fn`이 원시 학습 데이터에 적용됩니다. 이때 계산된 통계를 사용하여 인스턴스 수준 방식으로 데이터 레코드를 처리합니다(예: 숫자 열을 조정하기 위해).

이러한 2 단계 접근 방식을 통해 전체 전달 변환의 [사전 처리 과제](#)를 해결할 수 있습니다.

평가 데이터를 사전 처리하기 위해 `transform_fn`의 논리와 학습 데이터의 분석 단계에서 계산된 통계를 사용하여 인스턴스 수준 작업만 적용합니다. 다시 말해, μ 와 σ 같은 새로운 통계를 계산하여 평가 데이터에서 숫자 특성을 정규화하기 위해 전체 전달 방식으로 평가 데이터를 분석하지 않습니다. 대신 학습 데이터에서 계산된 통계를 사용하여 평가 데이터를 인스턴스 수준 방식으로 변환합니다.

변환된 학습 및 평가 데이터는 모델을 학습시키는 데 사용되기 전에 **Dataflow**를 사용하여 규모에 맞춰 준비됩니다. 이 일괄 데이터 준비 프로세스는 데이터를 미리 준비하여 학습 효율을 높이는 [사전 처리 과제](#)를 해결합니다. 그림 4와 같이 모델 내부 인터페이스는 변환된 특성을 예상합니다.

내보낸 모델에 변환 연결

위에서 설명한 것처럼 `transform_fn(tf.Transform` 파이프라인에서 생성)은 내보낸 **TensorFlow** 그래프로 저장됩니다. 이 그래프는 변환 논리를 인스턴스 수준 작업으로 사용하고, 전체 전달 변환에서 계산된 모든 통계를 그래프 상수로 사용합니다. 학습시킨 모델을 내보내 제공할 때는 `transform_fn` 그래프가 `serving_input_fn`의 일부로 `SavedModel`에 연결됩니다.

예측을 위해 모델을 제공하는 동안 모델 제공 인터페이스는 원시 형식(즉, 변환 이전)의 데이터 포인트를 예상합니다. 하지만 모델 내부 인터페이스는 변환된 형식의 데이터를 예상합니다.

이제 모델의 일부분인 `transform_fn`은 들어오는 데이터 포인트에서 모든 사전 처리 논리를 적용합니다. 저장된 상수(예: 숫자 특성을 정규화하기 위한 μ 및 σ)를 예측 중에 인스턴스 수준 작업에 사용합니다. 따라서 `transform_fn`은 원시 데이터 포인트를 그림 4와 같이 예측을 생성하기 위해 모델 내부 인터페이스가 예상하는 변환된 형식으로 전환합니다.

이는 학습/제공 편향의 [사전 처리 과제](#)를 해결합니다. 학습 및 평가 데이터를 변환하는 데 사용되는 것과 동일한 논리(구현)가 예측 제공 중에 적용되어 새로운 데이터 포인트를 변환하기 때문입니다.

사전 처리 옵션 요약

다음 표에는 이 문서에서 설명한 데이터 사전 처리 옵션이 요약되어 있습니다. 이 표는 다음과 같이 정리되어 있습니다.

- 행은 변환을 구현하는 데 사용할 수 있는 도구를 나타냅니다.
- 열은 세부사항을 기준으로 변환 유형을 나타냅니다.
- 하위 열(예: **일괄 점수 매기기**)은 모델 제공 요구사항을 나타냅니다.
- 각 셀은 제공 요구사항(하위 열)과 관련하여 변환 유형(열)을 구현하는 데 사용할 권장 도구(행)를 나타냅니다.

	인스턴스 수준 (스테이트리스(Stateless) 변환)		학습 도중 전체 전달, 제공 도중 인스턴스 수준(스테이트풀(Stateful) 변환)		학습 및 제공 도중 실시간(기간) 집계(스트리밍 변환)	
	일괄 점수 매기기	온라인 예측	일괄 점수 매기기	온라 인 예측	일괄 점수 매기기	온라인 예측
BigQuery(SQL)	양호 학습 및 일괄 점수 매기기 도중에 동일한 변환 구현이 데이터에 적용됩니다.	학습 데이터 처리는 가능 하지만 권장되지 않습니다. 서로 다른 도구를 사용하여 제공 데이터를 처리하기 때문에 학습/제공 편향이 발생합니다.	인스턴스 수준 일괄/온라인 변환에 BigQuery 를 사용하여 계산된 통계를 사용하는 것이 가능 합니다. 쉽지 않습니다. 통계 저장소가 학습 도중에 채워지고 예측 도중에 사용되도록 유지해야 합니다.		해당 없음 실시간 이벤트 를 기준으 로 계산된 통계를 집계합 니다.	학습 데이터를 처리하는 것이 가능 하지만 추천하지 않습니다. 서로 다른 도구를 사용하여 제공 데이터를 처리하기 때문에 학습/제공 편향이 발생합니다.
Dataflow(Apache Beam)		제공 시점의 데이터가 Dataflow 가 사용할 Pub/Sub 에 가져온 것이면 양호 합니다. 그 외의 경우에는 학습/제공 편향이 발생합니다.	인스턴스 수준 일괄/온라인 변환에 Dataflow 를 사용하여 계산된 통계를 사용하는 것이 가능 합니다. 쉽지 않습니다. 통계 저장소가 학습 도중에			양호 학습(일괄) 및 제공(스트림) 도중에 동일한 Apache Beam 변환이 데이터에 적용됩니다.

			채워지고 예측 도중에 사용되도록 유지해야 합니다.		
Dataflow (Apache Beam + TFT)	양호 학습 및 일괄 점수 매기기 도중에 동일한 변환 구현이 데이터에 적용됩니다.	추천 학습/제공 편향을 방지하고 학습 데이터를 미리 준비합니다.	추천 학습 도중의 변환 논리 + 계산된 통계가 제공을 위해 내보낸 모델에 연결된 <code>tf.Graph</code> 로 저장됩니다.		
TensorFlow* (<code>input_fn</code> 및 <code>serving_input_fn</code>)	가능하지만 추천하지 않음 학습 및 예측 효율성을 위해 학습 데이터를 미리 준비하는 것이 낫습니다.	가능하지만 추천하지 않음 학습 효율성을 위해 학습 데이터를 미리 준비하는 것이 낫습니다.	불가능		불가능

* 교차, 임베딩, 원-핫 인코딩 같은 변환은 `feature_columns` 로 선언적으로 수행되어야 합니다.

이 문서에서는 [TensorFlow Transform](#)(`tf.Transform`)을 사용하여 머신러닝(ML)을 위한 데이터 사전 처리를 구현하는 방법을 설명합니다. `tf.Transform`은 데이터 사전 처리 파이프라인을 통해 인스턴스 수준 및 전체 전달 데이터 변환을 모두 정의할 수 있는 TensorFlow 용 라이브러리입니다. 이러한 파이프라인은 [Apache Beam](#)을 통해 효율적으로 실행되며 TensorFlow 그래프가 부산물로 생성되어 모델이 제공될 때와 동일한 예측 도중 변환을 적용합니다.

이 문서에서는 [Dataflow](#)를 Apache Beam 실행자로 사용하여 각 단계를 표시하는 엔드 투 엔드 예시를 살펴봅니다. 여기서는 사용자가 [BigQuery](#), [Dataflow](#), [AI Platform\(ML\) Engine](#), TensorFlow [Estimator](#) API에 익숙하다고 가정합니다.

소개

[TensorFlow Transform](#)(`tf.Transform`)은 전체 전달이 필요한 변환에 유용한 TensorFlow 로 데이터를 사전 처리하는 라이브러리입니다. `tf.Transform` 출력은 TensorFlow 그래프 형식으로 내보내지며 이 그래프는 학습 및 제공에 사용할 인스턴스 수준 변환 논리를 전체 전달 변환에서 계산된 통계와 함께 표시합니다. 학습 및 제공에 같은 그래프를 사용하면 동일한 변환이 두 단계에 모두 적용되기 때문에 편향을 방지할 수 있습니다. 또한 `tf.Transform`은 Dataflow 의 일괄 처리 파이프라인에서 대규모로 실행되어 학습 데이터를 미리 준비하고 학습 효율을 높일 수 있습니다.

GCP 에서 사전 처리 유형, 과제, 옵션의 개념에 대해 자세히 알아보려면 [TensorFlow Transform 을 사용한 머신러닝을 위한 데이터 사전 처리 - 1부](#)를 참조하세요.

그림 1 은 `tf.Transform` 동작이 학습 및 예측용 데이터를 사전 처리하고 변환하는 개념을 개략적으로 보여줍니다.

그림 1. `tf.Transform`의 동작

[1부](#)에서 다루었듯이 `tf.Transform`은 다음과 같은 방식으로 사용됩니다.

1. `tf.Transform`에 구현된 변환을 사용하고 Dataflow 에서 이를 대규모로 실행하여 원시 학습 데이터를 사전 처리합니다. 사전 처리는 분석과 변환이라는 두 단계로 진행됩니다.
2. 분석 단계 중에는 상태 저장 변환의 필수 통계(예: 평균 및 편차)가 학습 데이터와 관련된 전체 전달 작업과 함께 계산됩니다. 변환 단계에서 이러한 계산된 통계는 인스턴스 수준 작업으로 학습 데이터(예: 숫자 특성의 z 점수 정규화)를 처리하는 데 사용됩니다.
3. `tf.Transform` 사전 처리 파이프라인은 변환된 학습 데이터와 `transform_fn` 함수라는 두 가지 주요 출력을 생성합니다.
 - 변환된 학습 데이터는 모델을 학습시키는 데 사용됩니다. 이때 모드 인터페이스는 변환된 특성을 예상합니다.
 - `transform_fn`은 TensorFlow 그래프 형식의 변환 논리로 구성됩니다. 여기에서 변환은 인스턴스 수준 작업이며, 전체 전달 변환에서 계산된 통계는 상수입니다.
4. 학습 후에 모델을 내보낼 때, 내보내기 프로세스는 `transform_fn` 그래프를 내보낸 `SavedModel`에 연결합니다. 예측을 위해 모델을 서빙하는 동안 모델 서빙 인터페이스는 원시 형식의 데이터 포인트를 예상합니다. 이제 모델의 일부분인 `transform_fn`은 들어오는 데이터 포인트에서 모든 사전 처리 논리를 적용합니다. 또한, 평균과 편차 같은 저장된 상수를 사용하여 예측 중에 숫자 특성을 인스턴스 수준 작업으로 정규화합니다.
5. `transform_fn`은 원시 데이터 포인트를 예측을 생성하기 위해 모델 인터페이스가 예상하는 변환된 형식으로 전환합니다.

이 솔루션의 Jupyter 메모장

구현 예시를 보여주는 Jupyter 메모장 두 개가 있으며, 둘 다 [GitHub](#) 저장소에 있습니다.

- [메모장 #1](#)은 데이터 사전 처리 부분을 다룹니다. 자세한 내용은 [Apache Beam 파이프라인 구현](#) 섹션에서 설명합니다.
- [메모장 #2](#)는 모델 학습 부분을 다룹니다. 자세한 내용은 [TensorFlow 모델 구현](#) 섹션에서 설명합니다.

Apache Beam 파이프라인 구현

이 섹션에서는 `tf.Transform`을 사용하여 데이터를 사전 처리하는 방법을 실제 예시를 통해 설명합니다. 이 도움말에서는 다양한 입력을 기준으로 아기 체중을 예측하는 데 사용되는 **Natality** 데이터 세트를 사용합니다. 데이터는 **BigQuery**의 [natality](#) 테이블에 저장됩니다.

예시에서 **Dataflow**는 `tf.Transform` 파이프라인을 규모에 맞게 실행하여 데이터를 준비하고 `transform_fn` 아티팩트를 만듭니다. 그다음의 코드는 파이프라인을 실행합니다. 이 문서의 이후 섹션에서는 파이프라인에서 각 단계를 수행하는 함수에 대해 설명합니다. 전체적인 파이프라인 단계는 다음과 같습니다.

1. **BigQuery**에서 학습 데이터를 읽습니다.
2. `tf.Transform`을 사용하여 학습 데이터를 분석하고 변환합니다.
3. 변환된 학습 데이터를 **Cloud Storage**에 `tfRecords`로 씁니다.
4. **BigQuery**에서 평가 데이터를 읽습니다.
5. 2 단계에서 생성된 `transform_fn`을 사용하여 평가 데이터를 변환합니다.
6. 변환된 학습 데이터를 **Cloud Storage**에 `tfrecords`로 씁니다.
7. 모델을 만들고 내보내기 위해 변환 아티팩트를 **Cloud Storage**에 씁니다.

다음 목록은 전체 파이프라인의 **Python** 코드를 보여줍니다. 각 단계에 대한 설명과 코드 목록은 이후 섹션에 나와 있습니다.

```
def run_transformation_pipeline(args):  
  
    pipeline_options = beam.pipeline.PipelineOptions(flags=[], **args)  
  
    runner = args['runner']
```

```

data_size = args['data_size']
transformed_data_location = args['transformed_data_location']
transform_artefact_location = args['transform_artefact_location']
temporary_dir = args['temporary_dir']
debug = args['debug']

# Instantiate the pipeline
with beam.Pipeline(runner, options=pipeline_options) as pipeline:
    with impl.Context(temporary_dir):

        # Preprocess train data
        step = 'train'
        # Read raw train data from BigQuery
        raw_train_dataset = read_from_bq(pipeline, step, data_size)
        # Analyze and transform raw_train_dataset
        transformed_train_dataset, transform_fn =
analyze_and_transform(raw_train_dataset, step)
        # Write transformed train data to sink as tfrecords
        write_tfrecords(transformed_train_dataset,
transformed_data_location, step)

        # Preprocess evaluation data
        step = 'eval'
        # Read raw eval data from BigQuery
        raw_eval_dataset = read_from_bq(pipeline, step, data_size)
        # Transform eval data based on produced transform_fn
        transformed_eval_dataset = transform(raw_eval_dataset,
transform_fn, step)
        # Write transformed eval data to sink as tfrecords
        write_tfrecords(transformed_eval_dataset,
transformed_data_location, step)

        # Write transformation artefacts
        write_transform_artefacts(transform_fn,
transform_artefact_location)

        # (Optional) for debugging, write transformed data as text
        step = 'debug'
        # Wwrite transformed train data as text if debug enabled
        if debug == True:
            write_text(transformed_train_dataset,
transformed_data_location, step)

```


1 단계: BigQuery 에서 원시 학습 데이터 읽기

첫 번째 단계는 `read_from_bq` 메서드를 사용하여 **BigQuery** 에서 원시 학습 데이터를 읽는 것입니다. 이 메서드는 **BigQuery** 에서 추출된 `raw_dataset` 객체를 반환합니다. `data_size` 값과 `step` 의 값(`train` 또는 `eval`)을 전달합니다. **BigQuery** 소스 쿼리는 `get_source_query` 메서드를 사용하여 생성됩니다.

```
def read_from_bq(pipeline, step, data_size):

    source_query = get_source_query(step, data_size)
    raw_data = (
        pipeline
        | '{} - Read Data from BigQuery'.format(step) >> beam.io.Read(
            beam.io.BigQuerySource(query=source_query,
use_standard_sql=True))
        | '{} - Clean up Data'.format(step) >> beam.Map(prepare_bq_row)
    )

    raw_metadata = create_raw_metadata()
    raw_dataset = (raw_data, raw_metadata)
    return raw_dataset
```

`tf.Transform` 사전 처리를 수행하기 전에 `Map`, `Filter`, `Group`, `Window` 등 일반적인 **Apache Beam** 기반 처리를 수행해야 할 수도 있습니다. 이 예시에서 코드는 `beam.Map(prepare_bq_row)`를 사용하여 **BigQuery** 에서 읽은 레코드를 정리합니다. 여기서 `prepare_bq_row`는 커스텀 메서드입니다. 이 커스텀 메서드는 카테고리 특성의 숫자 코드를 사람이 읽을 수 있는 라벨로 변환합니다.

또한 `tf.Transform`을 사용하여 **BigQuery** 에서 추출된 `raw_data` 객체를 분석 및 변환하려면 `(raw_data, raw_metadata)`의 튜플인 `raw_dataset` 객체를 만들어야 합니다. `raw_metadata` 객체는 다음과 같이 `create_raw_metadata` 메서드를 사용하여 생성됩니다.

```
CATEGORICAL_FEATURE_NAMES = ['is_male', 'mother_race']
NUMERIC_FEATURE_NAMES = ['mother_age', 'plurality', 'gestation_weeks']
TARGET_FEATURE_NAME = 'weight_pounds'
KEY_COLUMN = 'key'

def create_raw_metadata():

    raw_data_schema = {}

    # key feature schema
    raw_data_schema[KEY_COLUMN] = dataset_schema.ColumnSchema(
```

```

        tf.float32, [], dataset_schema.FixedColumnRepresentation())

# target feature schema
raw_data_schema[TARGET_FEATURE_NAME]= dataset_schema.ColumnSchema(
    tf.float32, [], dataset_schema.FixedColumnRepresentation())

# categorical feature schema
raw_data_schema.update({ column_name : dataset_schema.ColumnSchema(
    tf.string, [], dataset_schema.FixedColumnRepresentation())
    for column_name in CATEGORICAL_FEATURE_NAMES})

# numerical feature schema
raw_data_schema.update({ column_name : dataset_schema.ColumnSchema(
    tf.float32, [], dataset_schema.FixedColumnRepresentation())
    for column_name in NUMERIC_FEATURE_NAMES})

# create dataset_metadata given raw_data_schema
raw_metadata = dataset_metadata.DatasetMetadata(
    dataset_schema.Schema(raw_data_schema))

return raw_metadata

```

이 `raw_metadata` 객체는 `seving_input_receiver_fn`이 내보낸 모델 서빙 인터페이스를 만드는 데도 사용됩니다. 이 인터페이스는 그림 1 과 같이 원시 형식의 데이터 포인트를 예상합니다. 예측을 제공하기 위해 모델을 내보내는 작업은 뒷부분의 [예측을 제공하기 위해 모델 내보내기](#)에서 설명합니다.

이 메서드를 정의하는 메모장에서 셀을 실행하는 경우에는 `raw_metadata.schema`의 콘텐츠가 표시됩니다. 다음과 같은 열을 포함합니다.

- `mother_race(tf.string)`
- `weight_pounds(tf.float32)`
- `gestation_weeks(tf.float32)`
- `key(tf.float32)`
- `is_male(tf.string)`
- `mother_age(tf.string)`

2 단계: `preprocess_fn` 을 사용하여 원시 학습 데이터 변환

ML 을 위한 학습 데이터를 준비하기 위해 학습 데이터의 입력 원시 특성에 일반 사전 처리 변환을 적용하려고 한다고 가정해 보겠습니다. 이러한 변환에는 전체 전달 및 인스턴스 수준 작업이 포함됩니다. 특히, 다음 표에 나열된 변환을 수행하려고 합니다.

입력 특성	변환	필요한 통계	유형	출력 특성
weight_pound	없음	없음	해당 없음	weight_pound
mother_age	정규화	mean, var	전체 전달	mother_age_normalized
mother_age	동일 크기 버킷화	quantiles	전체 전달	mother_age_bucketized
mother_age	로그 계산	없음	인스턴스 수준	mother_age_log
plurality	아기가 한 명인지 여러 명인지를 나타냄	없음	인스턴스 수준	is_multiple
is_multiple	명목값을 숫자 색인으로 변환	vocab	전체 전달	is_multiple_index
gestation_weeks	0에서 1 사이	min, max	전체 전달	gestation_weeks_scaled
mother_race	명목값을 숫자 색인으로 변환	vocab	전체 전달	mother_race_index
is_male	명목값을 숫자 색인으로 변환	vocab	전체 전달	is_male_index

이러한 변환은 텐서 사전(input_features)을 예상하고 처리된 특성 사전(output_features)을 반환하는 preprocess_fn 메서드에서 구현됩니다.

다음 코드에서는 preprocess_fn 메서드를 tf.Transform 전체 전달 변환 API(tft. 프리픽스 포함) 및 TensorFlow(tf. 프리픽스 포함) 인스턴스 수준 작업을 사용하여 구현하는 방법을 보여줍니다.

```
def preprocess_fn(input_features):

    output_features = {}

    # target feature
    output_features['weight_pounds'] = input_features['weight_pounds']

    # normalization
    output_features['mother_age_normalized'] =
tft.scale_to_z_score(input_features['mother_age'])
    output_features['gestation_weeks_normalized'] =
tft.scale_to_z_score(input_features['gestation_weeks'])

    # bucketization based on quantiles
    output_features['mother_age_bucketized'] =
tft.bucketize(input_features['mother_age'], num_buckets=5)

    # you can compute new features based on custom formulas
    output_features['mother_age_log'] =
tft.log(input_features['mother_age'])

    # or create flags/indicators
    is_multiple = tf.as_string(input_features['plurality'] >
tft.constant(1.0))

    # convert categorical features to indexed vocab
    output_features['mother_race_index'] =
tft.compute_and_apply_vocabulary(input_features['mother_race'],
    vocab_filename='mother_race')
    output_features['is_male_index'] =
tft.compute_and_apply_vocabulary(input_features['is_male'],
    vocab_filename='is_male')
    output_features['is_multiple_index'] =
tft.compute_and_apply_vocabulary(is_multiple,
    vocab_filename='is_male')
    return output_features
```

`tf.Transform` [프레임워크](#)에는 앞의 예시와 같은 변환 이외에도 다음 표에 나열된 변환을 포함하여 여러 가지 다른 변환이 있습니다.

변환	적용 대상	설명
<code>scale_by_min_max</code>	숫자 특성	숫자 열을 <code>[output_min, output_max]</code> 범위로 조정합니다.
<code>scale_to_0_1</code>	숫자 특성	<code>[0,1]</code> 범위가 되도록 조정된 입력 열을 반환합니다.
<code>scale_to_z_score</code>	숫자 특성	평균이 0이고 편차가 1인 표준화된 열을 반환합니다.
<code>tfidf</code>	텍스트 특성	<code>x</code> 에 있는 용어를 용어 빈도 * 역문서 빈도에 매핑합니다.
<code>compute_and_apply_vocabulary</code>	카테고리 특성	카테고리 특성의 어휘를 생성하고 이 어휘가 있는 정수에 매핑합니다.
<code>ngrams</code>	텍스트 특성	N-그램의 <code>SparseTensor</code> 를 만듭니다.
<code>hash_strings</code>	범주형 특성	문자열을 버킷에 해시합니다.
<code>pca</code>	숫자 특성	편향된 공분산을 사용하여 데이터셋에서 PCA를 계산합니다.
<code>bucketize</code>	숫자 특성	동일 크기(백분위수 기준)의 버킷화된 열을 각 입력에 할당된 버킷 색인과 함께 반환합니다.

변환	적용 대상	설명
----	-------	----

니다.

`preprocess_fn` 메서드에서 구현된 변환을 파이프라인 이전 단계에서 생성된 `raw_train_dataset` 객체에 적용하려면 `AnalyzeAndTransformDataset` 메서드를 사용합니다. 이 메서드는 `raw_dataset` 객체를 입력으로 예상하고, `preprocess_fn` 메서드를 적용하고, `transformed_dataset` 객체 및 `transform_fn`을 생성합니다. 다음 코드는 이 처리를 설명합니다.

```
def analyze_and_transform(raw_dataset, step):

    transformed_dataset, transform_fn = (
        raw_dataset
        | '{} - Analyze & Transform'.format(step) >>
        impl.AnalyzeAndTransformDataset(preprocess_fn)
    )

    return transformed_dataset, transform_fn
```

변환은 분석 단계와 변환 단계의 두 가지 단계에서 원시 데이터에 적용됩니다. 이 문서 후반부에 있는 그림 5에서는 `AnalyzeAndTransformDataset`가 어떻게 `AnalyzeDataset` 및 `TransformDataset`로 분할되는지 보여줍니다.

분석 단계

분석 단계에서는 변환에 필요한 통계를 계산하기 위해 원시 학습 데이터를 전체 전달 프로세스에서 분석합니다. 여기에는 평균, 편차, 최솟값, 최댓값, 수량, 어휘 계산이 포함됩니다. 분석 프로세스에는 원시 데이터 세트(원시 데이터 + 원시 메타데이터)가 필요하며 다음 두 가지 출력이 생성됩니다.

- `transform_fn`. 분석 단계에서 계산된 통계 및 변환 논리(통계 사용)를 인스턴스 수준 작업으로 포함하는 함수입니다. 후반부의 [7 단계](#) 설명처럼 `transform_fn`은 저장되어 모델 `serving_input_fn`에 연결됩니다. 따라서 동일한 변환을 온라인 예측 데이터 포인트에 적용할 수 있습니다.
- `transform_metadata`. 변환 후 예상되는 데이터 스키마를 설명하는 객체입니다.

분석 단계는 그림 2에 설명되어 있습니다.

그림 2. tf.Transform 분석 단계

tf.Transform [분석기](#)는 min, max, sum, size, mean, var, covariance, quantiles, vocabulary, pca 를 포함합니다.

변환 단계

변환 단계에서는 변환된 학습 데이터를 생성하기 위해 분석 단계에서 생성된 transform_fn 을 사용하여 인스턴스 수준 프로세스에서 원시 학습 데이터를 변환합니다. 변환된 학습 데이터는 변환된 메타데이터(분석 단계에서 생성됨)와 페어링되어 transformed_train_dataset 를 생성합니다.

변환 단계는 그림 3 에 설명되어 있습니다.

그림 3. tf.Transform 변환 단계

특성을 사전 처리하려면 preprocess_fn 을 구현할 때 필수 tensorflow_transform(코드에서 tft 로 가져옴) 변환을 호출합니다. 예를 들어 tft.scale_to_z_score 를 호출하면 tf.Transform 이 이 함수 호출을 평균 및 편차 분석기로 번역하고 통계를 계산한 후 이 통계를 적용하여 숫자 특성을 정규화합니다. 이 작업은 다음 메서드를 호출할 때 자동으로 수행됩니다.

```
AnalyzeAndTransformDataset(preprocess_fn)
```

이 호출에 의해 생성되는 transformed_metadata.schema 항목은 다음 열을 포함합니다.

- gestation_weeks_normalixed(tf.float32)
- is_multiple_index(tf.int64, _is_categorical: True)
- mother_race_index(tf.int64, _is_categorical: True)
- is_male_index(tf.int64, _is_categorical: True)
- mother_age_log(tf.float32)
- mother_age_bucketized(tf.int64, _is_categorical: True)
- mother_age_normalized(tf.float32)
- weight_pounds(tf.float32)

3 단계: 변환된 학습 데이터 쓰기

학습 데이터가 분석 및 변환 단계를 통해 `tf.Transform` `preprocess_fn` 으로 사전 처리되면 **TensorFlow** 모델을 학습시키는 데 사용할 싱크에 데이터를 쓸 수 있습니다. **Dataflow** 를 사용하여 **Apache Beam** 파이프라인을 실행할 때 싱크는 **Cloud Storage** 입니다. 그 외의 경우 싱크는 로컬 디스크입니다. 데이터를 **CSV** 파일이나 고정 너비 형식의 파일로 작성할 수 있지만, **TensorFlow** 데이터 세트의 권장 파일 형식은 **TFRecord** 형식입니다. 이 형식은 `tf.train.Example` 프로토콜 버퍼 메시지로 구성된 간단한 레코드 기반 바이너리 형식입니다.

각 `tf.train.Example` 레코드에는 특성이 한 개 이상 포함됩니다. 이러한 특성은 학습을 위해 모델에 공급될 때 텐서로 변환됩니다. 다음 코드는 변환된 데이터세트를 지정된 위치에 있는 **TFRecord** 파일에 씁니다.

```
def write_tfrecords(dataset, location, step):

    transformed_data, transformed_metadata = transformed_dataset
    (
        transformed_data
        | '{} - Write Transformed Data'.format(step) >>
        beam.io.tfrecordio.WriteToTFRecord(
            file_path_prefix=os.path.join(location, '{}-'.format(step)),
            file_name_suffix=".tfrecords",

            coder=example_proto_coder.ExampleProtoCoder(transformed_metadata.schema)
        )
    )
```

4, 5, 6 단계: 평가 데이터 읽기, 변환, 쓰기

학습 데이터를 변환하고 `transform_fn` 함수를 생성한 후에는 이 함수를 사용하여 평가 데이터를 변환할 수 있습니다. 먼저 [1 단계: BigQuery 에서 원시 학습 데이터 읽기](#)에서 설명한 `read_from_bq` 메서드를 사용하여 **BigQuery** 에서 평가 데이터를 정리하고 `step` 매개변수의 `eval` 을 전달합니다.

다음으로 다음 코드를 사용하여 원시 평가 데이터 세트(`raw_dataset`)를 아래 코드와 같이 예상 변환 형식(`transformed_dataset`)으로 변환합니다.

```
def transform(raw_dataset, transform_fn, step):
    transformed_dataset = (
        (raw_dataset, transform_fn)
        | '{} - Transform'.format(step) >> impl.TransformDataset()
    )
    return transformed_dataset
```

평가 데이터를 변환할 때는 인스턴스 수준 작업만 적용되며, `transform_fn` 에 있는 논리와 학습 데이터의 분석 단계에서 계산된 통계를 모두 사용합니다. 즉, 전체 전달 방식에서는 평가 데이터를 분석하여 평가 데이터의 숫자 특성 **z-점수**

정규화에 필요한 평균과 편차 등 새로운 통계를 계산하지 않습니다. 대신 학습 데이터에서 계산된 통계를 사용하여 평가 데이터를 인스턴스 수준 방식으로 변환합니다.

따라서 학습 데이터의 컨텍스트에서 `impl.AnalyzeAndTransform` 을 사용하여 통계를 계산하고 데이터를 변환합니다. 이와 동시에 평가 데이터를 변환하는 컨텍스트에서는 `impl.TransformDataset` 를 사용하여 학습 데이터에서 계산된 통계를 사용하는 데이터만 변환합니다.

마지막으로, 학습 프로세스 중에 **TensorFlow** 모델을 평가하기 위해 데이터를 싱크(실행자에 따라 **Cloud Storage** 또는 로컬 디스크)에 **TFRecord** 파일로 작성합니다. 이렇게 하려면 [3 단계: 변환된 학습 데이터 쓰기](#)에서 설명한 `write_tfrecords` 메서드를 사용합니다. 그림 4에서는 학습 데이터의 분석 단계에서 생성된 `transform_fn` 을 사용하여 평가 데이터를 변환하는 방법을 보여줍니다.

그림 4. `transform_fn`을 사용하여 평가 데이터 변환

7 단계: `transform_fn` 저장

`tf.Transform` 사전 처리 파이프라인의 마지막 단계는 학습 데이터의 분석 단계에서 생성된 아티팩트를 포함하는 `transform_fn` 을 저장하는 것입니다. `transform_fn` 을 저장하는 코드는 다음 `write_transform_artifacts` 메서드에 표시되어 있습니다.

```
def write_transform_artefacts(transform_fn, location):  
    (  
        transform_fn  
        | 'Write Transform Artifacts' >>  
        transform_fn_io.WriteTransformFn(location)  
    )
```

나중에 이러한 아티팩트는 서빙에 필요한 모델 학습과 내보내기에 사용됩니다. 후반부의 그림 6 처럼 다음과 같은 아티팩트도 생성됩니다.

- `saved_model.pb`. 이 아티팩트는 변환 논리가 포함된 **TensorFlow** 그래프를 나타냅니다. 이 그래프는 원시 데이터 포인트를 변환된 형식으로 변환하는 모델 서빙 인터페이스에 연결되어야 합니다.
- `variables`. 이 아티팩트는 학습 데이터의 분석 단계 중에 계산된 통계를 포함하며 `saved_model.pb` 의 변환 논리에 사용됩니다.
- `assets`. 이 아티팩트는 `compute_and_apply_vocabulary` 메서드로 처리되는 각 카테고리 특성마다 어휘 파일 한 개를 포함하며 입력 원시 공칭값을 숫자 색인으로 변환하기 위해 서빙 중에 사용됩니다.

- `transformed_metadata`. 이 아티팩트는 변환된 데이터의 스키마를 설명하는 `schema.json` 파일을 포함합니다.

Dataflow 에서 파이프라인 실행

Dataflow 에서 `tf.Transform` 파이프라인을 실행하려면 [메모장](#)에 있는 코드를 단계별로 실행합니다. [Datalab](#) 에서 메모장을 실행하는 방법에 대한 자세한 내용은 [구조화된 데이터를 사용한 머신러닝](#) 가이드를 참조하세요.

메모장의 첫 번째 코드 셀에서 다음과 같이 변경해야 합니다.

- `PROJECT` 변수를 프로젝트 이름으로 설정합니다.
- `BUCKET` 변수를 버킷 이름으로 설정합니다.
- `REGION` 변수를 버킷 리전으로 설정합니다.
- Dialogflow 에서 파이프라인을 실행할 수 있도록 `LOCAL_RUN` 변수를 `False` 로 설정합니다.

그림 5에서는 이 예시에서 설명하는 `tf.Transform` 파이프라인의 Dataflow 실행 그래프를 보여줍니다.

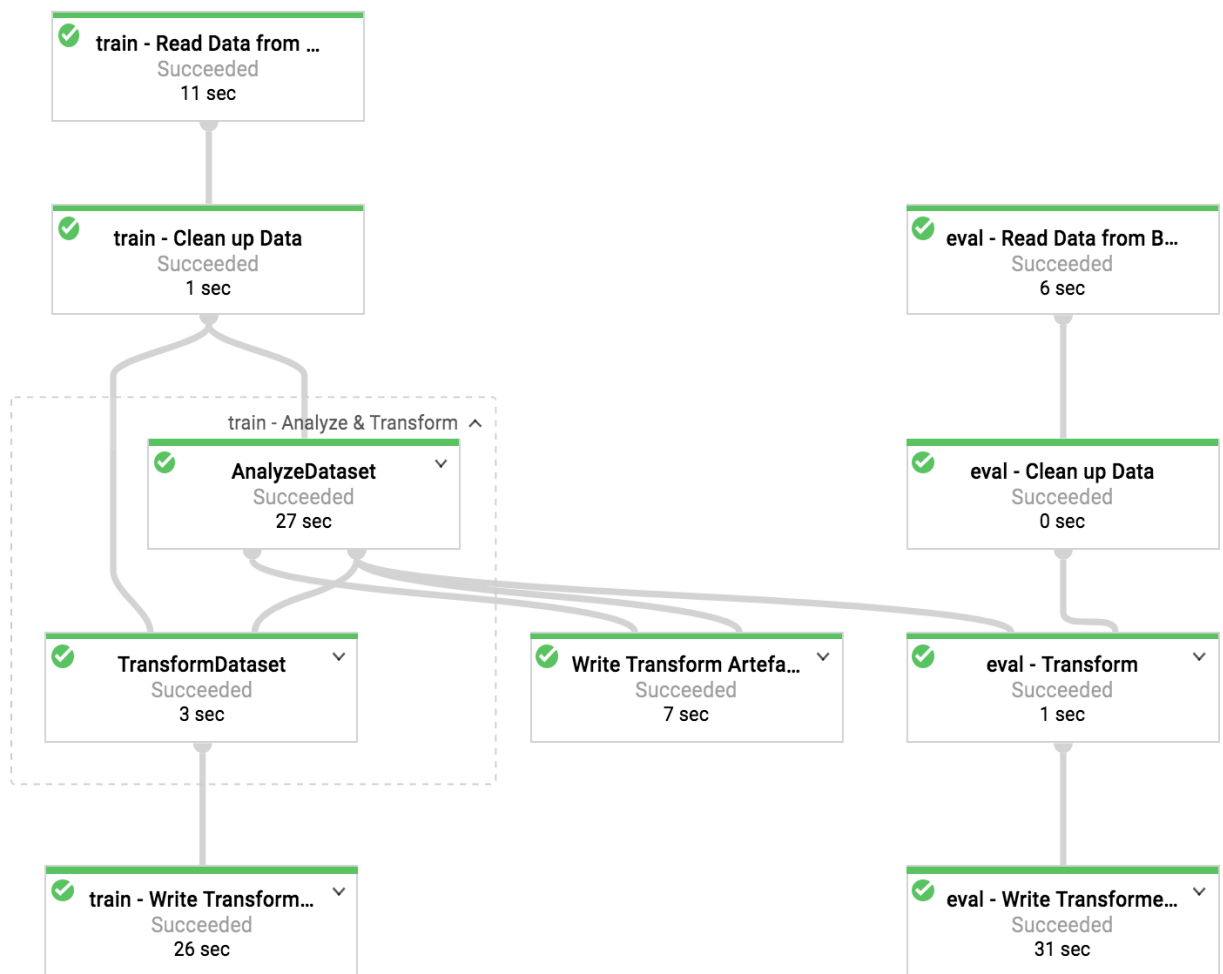


그림 5. `tf.Transform` 파이프라인의 Dataflow 실행 그래프

Dataflow 파이프라인을 실행하여 학습 및 평가 데이터를 사전 처리한 후에는 메모장에서 마지막 셀을 실행하여 **Cloud Storage**에서 생성된 객체를 살펴볼 수 있습니다. TFRecord 형식의 변환된 학습 및 평가 데이터는 다음 위치에 있습니다.

```
gs://[YOUR_BUCKET_NAME]/tft_babyweight/transformed
```

변환 아티팩트는 다음 위치에 생성됩니다.

```
gs://[YOUR_BUCKET_NAME]/tft_babyweight/transform
```

그림 6은 생성된 데이터 객체 및 아티팩트를 보여주는 파이프라인의 출력 목록입니다.

```

transformed data:
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/eval-00000-of-00001.tfrecords
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/train-00000-of-00003.tfrecords
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/train-00001-of-00003.tfrecords
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/train-00002-of-00003.tfrecords

transformed metadata:
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transformed_metadata/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transformed_metadata/v1-json/

transform artefact:
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/saved_model.pb
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/variables/

transform assets:
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/is_male
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/is_multiple
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/mother_race

```

그림 6. `tf.Transform` 파이프라인을 통한 출력으로 생성된 데이터 객체 및 아티팩트 목록

TensorFlow 모델 구현

아기 체중 추정 예시에서 TensorFlow 모델은 [DNNLinearCombinedRegressor](#) 를 사용하여 구현됩니다. 모델 생성, 학습, 평가, 내보내기를 위한 코드는 [GitHub](#) 저장소의 [메모장](#)입니다. 모델은 앞서 설명한 `tf.Transform` 사전 처리 파이프라인에서 생성한 데이터와 아티팩트를 사용합니다.

모델을 만들기 위한 단계는 다음과 같습니다.

1. `transform_metadata` 객체를 로드합니다.
2. `transform_metadata` 객체를 사용하여 학습 및 평가 데이터를 읽고 파싱하는 `input_fn` 을 만듭니다.
3. `transform_metadata` 객체를 사용하여 특성 열을 만듭니다.
4. `feature_columns` 를 사용하여 `DNNLinearCombinedRegressor` 에스티메이터를 만듭니다.
5. 에스티메이터를 학습시키고 평가합니다.
6. `transform_fn` 이 연결된 `serving_input_fn` 을 정의하여 에스티메이터를 내보냅니다.
7. 내보낸 모델을 [saved_model_cli](#) 도구를 사용하여 검사합니다.
8. 내보낸 모델을 예측에 사용합니다.

이 문서의 목표는 모델을 구축하는 것이 아닙니다. 따라서 모델을 만들고 학습시키는 방법에 대해서는 자세히 설명하지 않습니다. 하지만 이 섹션에서는 `transform_metadata` 객체(`tf.Transform` 프로세스에서 생성됨)를 사용하여 모델의 특성 열을 만드는 방법을 보여줍니다. 또한 서빙을 위해 모델을 내보낼 때 `transform_fn`(`tf.Transform` 프로세스에서 생성됨)을 `serving_input_fn`에서 사용하는 방법을 설명합니다.

생성된 변환 아티팩트를 모델 학습에 사용

TensorFlow 모델을 학습시킬 때는 이전 데이터 처리 단계에서 생성된 변환된 `train` 및 `eval` 객체를 사용합니다. 이러한 객체는 분할된 **TFRecord** 파일로 저장됩니다. 이전 단계에서 생성된 `transformed_metadata` 객체는 다음과 같은 용도에서 유용할 수 있습니다.

- 학습 및 평가를 위해 데이터(`tf.train.Example` 객체)를 파싱하여 모델에 공급합니다.
- 동적 메타데이터 기반 방식으로 특성 열을 만듭니다.

`tf.train.Example` 데이터 파싱

TFRecord 파일을 읽어 학습 및 평가 데이터를 모델에 공급하므로 파일에 있는 각 `tf.train.Example` 객체를 파싱하여 특성(텐서) 사전을 만들어야 합니다. 그러면 특성이 특성 열을 통해 모델 입력 레이어에 매핑되고, 이는 모델 학습/평가 인터페이스 역할을 수행합니다. 이전 단계에서 생성된 `transformed_metadata` 객체를 사용합니다. 이 과정은 두 단계로 구성됩니다.

먼저, [transform_fn 저장 섹션](#)의 설명대로 이전 사전 처리 단계에서 생성되고 저장된 `transformed_metadata` 객체를 로드합니다.

```
transformed_metadata = metadata_io.read_metadata(
    os.path.join(TRANSFORM_ARTEFACTS_DIR, "transformed_metadata"))
```

그런 다음 `transformed_metadata` 객체를 `feature_spec` 객체로 변환하고 `tfrecords_input_fn`에서 사용합니다.

```
def tfrecords_input_fn(files_name_pattern, transformed_metadata,
    mode=tf.estimator.ModeKeys.EVAL,
    num_epochs=1,
    batch_size=500):

    dataset = tf.contrib.data.make_batched_features_dataset(
        file_pattern=files_name_pattern,
```



```

        batch_size=batch_size,
        features=transformed_metadata.schema.as_feature_spec(),
        reader=tf.data.TFRecordDataset,
        num_epochs=num_epochs,
        shuffle=True if mode == tf.estimator.ModeKeys.TRAIN else False,
        shuffle_buffer_size=1+(batch_size*2),
        prefetch_buffer_size=1
    )

    iterator = dataset.make_one_shot_iterator()
    features = iterator.get_next()
    target = features.pop(TARGET_FEATURE_NAME)
    return features, target

```

특성 열 만들기

파이프라인은 모델에서 학습 및 평가를 수행하는 데 필요한 변환된 데이터의 스키마를 설명하는 `transformed_metadata` 객체를 생성합니다. 이 메타데이터를 사용하면 각각을 이름으로 지정하지 않고 특성 열을 동적으로 만들 수 있습니다. 이렇게 메타데이터 기반 동적 방식으로 특성 열을 만들면 특성이 수백 개 있을 때 유용합니다. 다음 코드에서는 메타데이터를 사용하여 특성 열을 만드는 방법을 보여줍니다.

```

def create_wide_and_deep_feature_columns(transformed_metadata,
                                          hparams):

    deep_feature_columns = []
    wide_feature_columns = []

    column_schemas = transformed_metadata.schema.column_schemas

    for feature_name in column_schemas:
        if feature_name == TARGET_FEATURE_NAME:
            continue

        # creating numerical features
        column_schema = column_schemas[feature_name]
        if isinstance(column_schema._domain,
dataset_schema.FloatDomain):
            deep_feature_columns.append(tf.feature_column.numeric_column(
feature_name))

        # creating categorical features with identity
        elif isinstance(column_schema._domain,

```

```

dataset_schema.IntDomain):
    if column_schema._domain._is_categorical==True:
        wide_feature_columns.append(
            tf.feature_column.categorical_column_with_identity(
                feature_name,
                num_buckets=column_schema._domain._max_value+1)
        )
    else:
        deep_feature_columns.append(tf.feature_column.numeric_column(feature_name))

    if hparams.extend_feature_columns==True:
        mother_race_X_mother_age_bucketized =
tf.feature_column.crossed_column(
    ['mother_age_bucketized', 'mother_race_index'], 55)

        wide_feature_columns.append(mother_race_X_mother_age_bucketized)

        mother_race_X_mother_age_bucketized_embedded =
tf.feature_column.embedding_column(
            mother_race_X_mother_age_bucketized,
hparams.embed_dimensions)
        deep_feature_columns.append(mother_race_X_mother_age_bucketized_embedded)

    return wide_feature_columns, deep_feature_columns

```

열 스키마가 FloatDomain 이면 이 코드는 `tf.feature_column.numeric_column` 열을 만듭니다. 반면에 열 스키마가 IntDomain 이고 `_is_categorical` 속성이 True 이면 `tf.feature_column.categorical_column_with_identity` 열을 만듭니다.

또한 [1부](#)의 '사전 처리를 수행할 위치' 섹션의 [옵션 C](#)에 설명된 확장 특성 열을 만들 수 있습니다. 이러한 문서에서 사용된 예시에서는 `tf.feature_column.crossed_column`을 통해 `mother_race` 특성과 `mother_age_bucketized` 특성을 교차하여 `mother_race_X_mother_age_bucketized`라는 새로운 특성을 만듭니다. 뿐만 아니라 `tf.feature_column.embedding_column` 특성 열을 통해 교차된 특성의 저차원 밀집 표현을 만듭니다.

그림 7에서는 변환된 데이터와 변환된 메타데이터를 사용하여 TensorFlow 에스티메이터를 정의하고 학습시키는 방법을 보여줍니다.

그림 7. 변환된 데이터로 TensorFlow 모델 학습

예측 서빙용 모델 내보내기

TensorFlow 모델(에스티메이터)을 학습시킨 후에는 예측을 위한 새로운 데이터 포인트를 서빙할 수 있도록 에스티메이터를 `SavedModel` 객체로 내보냅니다. 모델을 내보낼 때 해당 인터페이스, 즉 서빙 중에 예상되는 입력 특성 스키마를 정의해야 합니다. 이 입력 특성 스키마는 다음 코드에 나온 것처럼 `serving_input_fn`에서 정의됩니다.

```
def serving_input_fn():

    from tensorflow_transform.saved import saved_transform_io

    # get the feature_spec of raw data
    raw_metadata = create_raw_metadata()

    # create receiver placeholders to the raw input features
    raw_input_features = raw_metadata.schema.as_batched_placeholders()
    raw_input_features.pop(TARGET_FEATURE_NAME)
    raw_input_features.pop(KEY_COLUMN)

    # apply transform_fn on raw features
    _, transformed_features = (
        saved_transform_io.partially_apply_saved_transform(
            os.path.join(TRANSFORM_ARTEFACTS_DIR, transform_fn_io.TRANSFORM_F
N_DIR),
            raw_input_features)
    )

    return tf.estimator.export.ServingInputReceiver(
        transformed_features, raw_input_features)

export_dir = os.path.join(model_dir, 'export')

if tf.gfile.Exists(export_dir):
    tf.gfile.DeleteRecursively(export_dir)

estimator.export_savedmodel(
    export_dir_base=export_dir,
    serving_input_receiver_fn=serving_input_fn
)
```

서빙 중에 모델은 원시 형식의 데이터 포인트(변환 이전의 원시 특성)를 예상합니다. 따라서 `serving_input_fn`은 `raw_metadata` 객체를 사용하여 원시 특성의 수신자 자리표시자를 만듭니다. 하지만 앞서 설명한 대로 학습된 모델은

변환된 스키마의 데이터 포인트를 예상합니다. 따라서 원시 특성을 받은 후에는 저장된 `transform_fn` 을 자리표시자를 사용하여 받은 `raw_input_features` 객체에 적용하여 모델 인터페이스가 예상하는 `transformed_features` 로 변환합니다. 그림 8 은 서빙을 위해 모델을 내보내는 마지막 단계를 보여줍니다.

그림 8. `transform_fn`이 연결된 상태에서 서빙을 위해 모델 내보내기

예측용 모델 학습 및 사용

메모장의 셀을 실행하여 모델을 로컬에서 학습시킬 수 있습니다. AI Platform 을 사용하여 규모에 맞게 코드를 패키징하고 모델을 학습시키는 방법의 예시는 Google Cloud [cloudml-samples](#) GitHub 저장소의 샘플과 가이드를 참조하세요.

`saved_model_cli` 도구를 사용하여 내보낸 `SavedModel` 을 검사하면 그림 9 와 같이 `signature_def` 의 입력에 원시 특성이 포함된 것을 확인할 수 있습니다.

```
%%bash
```

```
saved_model_dir=${export_dir}/${ls ${export_dir} | tail -n 1}
echo ${saved_model_dir}
ls ${saved_model_dir}
saved_model_cli show --dir=${saved_model_dir} --all
```

```
babyweight_tft/models/dnn_estimator/export/1535488965
assets
saved_model.pb
variables
```

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

```
signature_def['predict']:
```

The given SavedModel SignatureDef contains the following input(s):

inputs['gestation_weeks'] tensor_info:

dtype: DT_FLOAT

shape: (-1)

name: gestation_weeks:0

inputs['is_male'] tensor_info:

dtype: DT_STRING

shape: (-1)

name: is_male:0

inputs['mother_age'] tensor_info:

dtype: DT_FLOAT

shape: (-1)

name: mother_age:0

inputs['mother_race'] tensor_info:

dtype: DT_STRING

shape: (-1)

name: mother_race:0

inputs['plurality'] tensor_info:

dtype: DT_FLOAT

shape: (-1)

name: plurality:0

The given SavedModel SignatureDef contains the following output(s):

outputs['predictions'] tensor_info:

dtype: DT_FLOAT

shape: (-1, 1)

name: add:0

Method name is: tensorflow/serving/predict

그

림 9. 내보낸 SavedModel 인터페이스

메모장의 마지막 셀에서는 내보낸 모델을 예측에 사용하는 방법을 보여줍니다. 입력(샘플) 데이터 포인트가 원시 스키마라는 점을 강조표시하는 것이 중요합니다. 온라인 예측용 모델을 마이크로서비스로 **AI Platform**에 배포하는 방법을 알아보려면 [모델 배포](#) 문서를 참조하세요.