aws / amazon-sagemaker-examples Public

Code Issues 525 Pull requests 80 Discussions Actions Projects Security Insights

master

amazon-sagemaker-examples / introduction_to_applying_machine_learning / US-census_population_segmentation_PCA_Kmeans / sagemaker-countycensusclustering.ipynb

 eitansela Update sagemaker-countycensusclustering.ipynb (#2840) History

7 contributors 

2770 lines (2770 sloc) | 80.5 KB

In [ ]:

```
# Install dependencies
import sys

!{sys.executable} -m pip install smdebug
!{sys.executable} -m pip install seaborn
!{sys.executable} -m pip install plotly
!{sys.executable} -m pip install opencv-python
!{sys.executable} -m pip install shap
!{sys.executable} -m pip install bokeh
!{sys.executable} -m pip install imageio
```

Analyze US census data for population segmentation using Amazon SageMaker

https://aws.amazon.com/blogs/machine-learning/analyze-us-census-data-for-population-segmentation-using-amazon-sagemaker/

Introduction

In the United States, with the 2018 midterm elections approaching, people are looking for more information about the voting process. This example notebook explores how we can apply machine learning (ML) to better integrate science into the task of understanding the electorate.

Typically, for machine learning applications, clear use cases are derived from labelled data. For example, based on the attributes of a device, such as its age or model number, we can predict its likelihood of failure. We call this *supervised learning* because there is supervision or guidance towards predicting specific outcomes.

However, in the real world, there are often large data sets where there is no particular outcome to predict, where clean labels are hard to define. It can be difficult to pinpoint exactly what the right outcome is to predict. This type of use case is often exploratory. It seeks to understand the makeup of a dataset and what natural patterns exist. This type of use case is known as `unsupervised learning` . One example of this is trying to group similar individuals together based on a set of attributes.

The use case this blog post explores is population segmentation. We have taken publicly available, anonymized data from the US census on demographics by different US counties: https://factfinder.census.gov/faces/nav/jsf/pages/index.xhtml. (Note that this

product uses the Census Bureau Data API but is not endorsed or certified by the Census Bureau.) The outcome of this analysis are natural groupings of similar counties in a transformed feature space. The cluster that a county belongs to can be leveraged to plan an election campaign, for example, to understand how to reach a group of similar counties by highlighting messages that resonate with that group. More generally, this technique can be applied by businesses in customer or user segmentation to create targeted marketing campaigns. This type of analysis has the ability to uncover similarities that may not be obvious at face value- such as counties CA-Fresno and AZ- Yuma County being grouped together. While intuitively they differ in commonly-examined attributes such as population size and racial makeup, they are more similar than different when viewed along axes such as the mix of employment type.

You can follow along in this sample notebook where you can run the code and interact with the data while reading through the blog post (link is shown above).

There are two goals for this exercise:

1) Walk through a data science workflow using Amazon SageMaker for `unsupervised learning` using `PCA` and `KMeans` modelling techniques.

2) Demonstrate how users can access the underlying models that are built within Amazon SageMaker to extract useful model attributes. Often, it can be difficult to draw conclusions from unsupervised learning, so being able to access the models for `PCA` and `KMeans` becomes even more important beyond simply generating predictions using the model.

The data science workflow has 4 main steps:

Step 1: Loading the data from Amazon S3

You need to load the dataset from an Amazon S3 bucket into the Amazon SageMaker notebook.

First, we will import the relevant libraries into our Amazon SageMaker notebook.

In [ ]:
```
import os
import boto3
import io
import sagemaker

%matplotlib inline

import pandas as pd
import numpy as np
import mxnet as mx
import matplotlib.pyplot as plt
import matplotlib
import seaborn as sns

matplotlib.style.use("ggplot")
import pickle, gzip, urllib, json
import csv
```

Amazon SageMaker integrates seamlessly with Amazon S3. During the first step in creating the notebook, we specified a `AmazonSageMakerFullAccess` role for the notebook. That gives this notebook permission to access any Amazon S3 bucket in this AWS account with "sagemaker" in its name.

The get_execution_role function retrieves the IAM role you created at the time you created your notebook instance.

In [ ]:
```
from sagemaker import get_execution_role

role = get_execution_role()
```

We can see our role is an AmazonSageMaker-ExecutionRole.

In [ ]:
```
role
```

Loading the dataset

Loading the dataset

I have previously downloaded and stored the data in a public S3 bucket that you can access. You can use the Python SDK to interact with AWS using a Boto3 client.

First, start the client.

In [ ]:
```
s3_client = boto3.client("s3")
data_bucket_name = "aws-ml-blog-sagemaker-census-segmentation"
```

You'll get a list of objects that are contained within the bucket. You can see there is one file in the bucket, 'Census_Data_for_SageMaker.csv'.

In [ ]:
```
obj_list = s3_client.list_objects(Bucket=data_bucket_name)
file = []
for contents in obj_list["Contents"]:
    file.append(contents["Key"])
print(file)
```

In [ ]:
```
file_data = file[0]
```

Grab the data from the CSV file in the bucket.

In [ ]:
```
response = s3_client.get_object(Bucket=data_bucket_name, Key=file_data)
response_body = response["Body"].read()
counties = pd.read_csv(io.BytesIO(response_body), header=0, delimiter=",", low_memory=False)
```

This is what the first 5 rows of our data looks like:

In [ ]:
```
counties.head()
```

Step 2: Exploratory data analysis *EDA* - Data cleaning and exploration

a. Cleaning the data

We can do simple data cleaning and processing right in our notebook instance, using `the compute` instance of the notebook to execute these computations

execute these computations.

How much data are we working with?

There are 3220 rows with 37 columns

In [ ]:
```
counties.shape
```

Let's just drop any incomplete data to make our analysis easier. We can see that we lost 2 rows of incomplete data, we now have 3218 rows in our data.

In [ ]:
```
counties.dropna(inplace=True)
counties.shape
```

Let's combine some descriptive reference columns such as state and county and leave the numerical feature columns.

We can now set the 'state-county' as the index and the rest of the numerical features become the attributes of each unique county.

In [ ]:
```
counties.index = counties["State"] + "-" + counties["County"]
counties.head()
drop = ["CensusId", "State", "County"]
counties.drop(drop, axis=1, inplace=True)
counties.head()
```

b. Visualizing the data

Now we have a dataset with a mix of numerical and categorical columns. We can visualize the data for some of our numerical columns and see what the distribution looks like.

In [ ]:
```
import seaborn as sns

for a in ["Professional", "Service", "Office"]:
    ax = plt.subplots(figsize=(6, 3))
    ax = sns.distplot(counties[a])
    title = "Histogram of " + a
    ax.set_title(title, fontsize=12)
    plt.show()
```

For example, from the figures above you can observe the distribution of counties that have a percentage of workers in Professional,

Service, or Office occupations. Viewing the histograms can visually indicate characteristics of these features such as the mean or skew. The distribution of Professional workers for example reveals that the typical county has around 25-30% Professional workers, with a right skew, long tail and a Professional worker % topping out at almost 80% in some counties.

c. Feature engineering

Data Scaling- We need to standardize the scaling of the numerical columns in order to use any distance based analytical methods so that we can compare the relative distances between different feature columns. We can use `minmaxscaler` to transform the numerical columns so that they also fall between 0 and 1.

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
counties_scaled = pd.DataFrame(scaler.fit_transform(counties))
counties_scaled.columns = counties.columns
counties_scaled.index = counties.index
```

We can see that all of our numerical columns now have a min of 0 and a max of 1.

In [ ]:
```
counties_scaled.describe()
```

Step 3: Data modelling

a. Dimensionality reduction

We will be using principal component analysis (PCA) to reduce the dimensionality of our data. This method decomposes the data matrix into features that are orthogonal with each other. The resultant orthogonal features are linear combinations of the original feature set. You can think of this method as taking many features and combining similar or redundant features together to form a new, smaller feature set.

We can reduce dimensionality with the built-in Amazon SageMaker algorithm for PCA.

We first import and call an instance of the PCA SageMaker model. Then we specify different parameters of the model. These can be resource configuration parameters, such as how many instances to use during training, or what type of instances to use. Or they can be model computation hyperparameters, such as how many components to use when performing PCA. Documentation on the PCA model can be found here: http://sagemaker.readthedocs.io/en/latest/pca.html

You will use the tools provided by the Amazon SageMaker Python SDK to upload the data to a default bucket.

In [ ]:
```
sess = sagemaker.Session()
bucket = sess.default_bucket()
```

In [ ]:
```
from sagemaker import PCA

num_components = 33

pca_SM = PCA(
    role=role,
    instance_count=1,
    instance_type="ml.c4.xlarge",
    output_path="s3://" + bucket + "/counties/",
    num_components=num_components,
)
```

Next, we prepare data for Amazon SageMaker by extracting the `numpy array` from the `DataFrame` and explicitly casting to `float32`

In [ ]:
```
train_data = counties_scaled.values.astype("float32")
```

The *record_set* function in the Amazon SageMaker `PCA` model converts a `numpy array` into a record set format that is the required format for the input data to be trained. This is a requirement for all Amazon SageMaker built-in models. The use of this data type is one of the reasons that allows training of models within Amazon SageMaker to perform faster, for larger data sets compared with other implementations of the same models, such as the `sklearn` implementation.

We call the fit function on our PCA model, passing in our training data, and this spins up a training instance or cluster to perform the training job.

In [ ]:
```
%%time
pca_SM.fit(pca_SM.record_set(train_data))
```

b. Accessing the PCA model attributes

After the model is created, we can also access the underlying model parameters.

Now that the training job is complete, you can find the job under **Jobs** in the **Training** subsection in the Amazon SageMaker console.

Model artifacts are stored in Amazon S3 after they have been trained. This is the same model artifact that is used to deploy a trained model using Amazon SageMaker. Since many of the Amazon SageMaker algorithms use MXNet for computational speed, the model artifact is stored as an ND array. For an output path that was specified during the training call, the model resides in `<training_job_name>/output/model.tar.gz` file, which is a TAR archive file compressed with GNU zip (`gzip`) compression.

In [ ]:
```
job_name = pca_SM.latest_training_job.name
model_key = "counties/" + job_name + "/output/model.tar.gz"

boto3.resource("s3").Bucket(bucket).download_file(model_key, "model.tar.gz")
os.system("tar -zxvf model.tar.gz")
```

After the model is decompressed, we can load the ND array using MXNet.

In [ ]:
```
import mxnet as mx

pca_model_params = mx.ndarray.load("model_algo-1")
```

Three groups of model parameters are contained within the PCA model.

`mean` is optional and is only available if the "subtract_mean" hyperparameter is true when calling the training step from the original PCA SageMaker function.

`v` contains the principal components (same as 'components_' in the `sklearn PCA` model).

`s` the singular values of the components for the PCA transformation. This does not exactly give the % variance from the original feature space, but can give the % variance from the projected feature space.

`explained-variance-ratio` ~= square(s) / sum(square(s))

To calculate the exact `explained-variance-ratio` vector if needed, it simply requires saving the sum of squares of the original data (call that N) and computing explained-variance-ratio = square(s) / N.

In [ ]:
```
s = pd.DataFrame(pca_model_params["s"].asnumpy())
v = pd.DataFrame(pca_model_params["v"].asnumpy())
```

We can now calculate the variance explained by the largest n components that we want to keep. For this example, let's take the top 5 components.

We can see that the largest 5 components explain ~72% of the total variance in our dataset:

In [ ]:
```
s.iloc[28:, :].apply(lambda x: x * x).sum() / s.apply(lambda x: x * x).sum()
```

After we have decided to keep the top 5 components, we can take only the 5 largest components from our original s and v matrix.

In [ ]:
```
s_5 = s.iloc[28:, :]
v_5 = v.iloc[:, 28:]
v_5.columns = [0, 1, 2, 3, 4]
```

We can now examine the makeup of each PCA component based on the weightings of the original features that are included in the component. For example, the following code shows the first component. We can see that this component describes an attribute of a county that has high poverty and unemployment, low income and income per capita, and high Hispanic/Black population and low White population.

Note that this is v_5[4] or last component of the list of components in v_5, but is actually the largest component because the components are ordered from smallest to largest. So v_5[0] would be the smallest component. Similarly, change the value of component_num to cycle through the makeup of each component.

In [ ]:
```
component_num = 1

first_comp = v_5[5 - component_num]
comps = pd.DataFrame(
    list(zip(first_comp, counties_scaled.columns)), columns=["weights", "features"]
)
comps["abs_weights"] = comps["weights"].apply(lambda x: np.abs(x))
ax = sns.barplot(
    data=comps.sort_values("abs_weights", ascending=False).head(10),
    x="weights",
    y="features",
    palette="Blues_d",
)
ax.set_title("PCA Component Makeup: #" + str(component_num))
plt.show()
```

Similarly, you can go through and examine the makeup of each PCA components and try to understand what the key positive and negative attributes are for each component. The following code names the components, but feel free to change them as you gain insight into the unique makeup of each component.

In [ ]:
```
PCA_list = ["comp_1", "comp_2", "comp_3", "comp_4", "comp_5"]

# PCA_list=["Poverty/Unemployment", "Self Employment/Public Workers", "High Income/Professional & Office Workers",
#         "Black/Native Am Populations & Public/Professional Workers", "Construction & Commuters"]
```

c. Deploying the PCA model

We can now deploy this model endpoint and use it to make predictions. This model is now live and hosted on an instance_type that we specify.

In [ ]:
```
%%time
pca_predictor = pca_SM.deploy(initial_instance_count=1, instance_type="ml.t2.medium")
```

We can also pass our original dataset to the model so that we can transform the data using the model we created. Then we can take the largest 5 components and this will reduce the dimensionality of our data from 34 to 5.

In [ ]:
```
%%time
result = pca_predictor.predict(train_data)
```

In [ ]:
```
counties_transformed = pd.DataFrame()
for a in result:
    b = a.label["projection"].float32_tensor.values
    counties_transformed = counties_transformed.append([list(b)])
counties_transformed.index = counties_scaled.index
counties_transformed = counties_transformed.iloc[:, 28:]
counties_transformed.columns = PCA_list
```

Now we have created a dataset where each county is described by the 5 principal components that we analyzed earlier. Each of these 5 components is a linear combination of the original feature space. We can interpret each of these 5 components by analyzing the makeup of the component shown previously.

In [ ]:
```
counties_transformed.head()
```

d. Population segmentation using unsupervised clustering

Now, we'll use the `KMeans` algorithm to segment the population of counties by the 5 `PCA` attributes we have created. `KMeans` is a clustering algorithm that identifies clusters of similar counties based on their attributes. Since we have ~3000 counties and 34 attributes in our original dataset, the large feature space may have made it difficult to cluster the counties effectively. Instead, we have reduced the feature space to 5 PCA components, and we'll cluster on this transformed dataset.

In [ ]:
```
train_data = counties_transformed.values.astype("float32")
```

First, we call and define the hyperparameters of our `KMeans` model as we have done with our `PCA` model. The `KMeans` algorithm allows the user to specify how many clusters to identify. In this instance, let's try to find the top 7 clusters from our dataset.

In [ ]:
```
from sagemaker import KMeans

num_clusters = 7
kmeans = KMeans(
    role=role,
    instance_count=1,
    instance_type="ml.c4.xlarge",
    output_path="s3://" + bucket + "/counties/",
    k=num_clusters,
)
```

Then we train the model on our training data.

In [ ]:
```
%%time
kmeans.fit(kmeans.record_set(train_data))
```

Now we deploy the model, and we can pass in the original training set to get the labels for each entry. This will give us which cluster each county belongs to.

In [ ]:
```
%%time
kmeans_predictor = kmeans.deploy(initial_instance_count=1, instance_type="ml.t2.medium")
```

In [ ]:
```
%%time
result = kmeans_predictor.predict(train_data)
```

```
result = kmeans_predictor.predict(train_data)
```

We can see the breakdown of cluster counts and the distribution of clusters.

In [ ]:
```
cluster_labels = [r.label["closest_cluster"].float32_tensor.values[0] for r in result]
```

In [ ]:
```
pd.DataFrame(cluster_labels)[0].value_counts()
```

In [ ]:
```
ax = plt.subplots(figsize=(6, 3))
ax = sns.distplot(cluster_labels, kde=False)
title = "Histogram of Cluster Counts"
ax.set_title(title, fontsize=12)
plt.show()
```

However, to improve `explainability`, we need to access the underlying model to get the cluster centers. These centers will help describe which features characterize each cluster.

Step 4: Drawing conclusions from our modelling

Explaining the result of the modelling is an important step in making use of our analysis. By combining `PCA` and `KMeans`, and the information contained in the model attributes within an Amazon SageMaker trained model, we can form concrete conclusions based on the data.

a. Accessing the KMeans model attributes

First, we will go into the bucket where the `KMeans` model is stored and extract it.

In [ ]:
```
job_name = kmeans.latest_training_job.name
model_key = "counties/" + job_name + "/output/model.tar.gz"

boto3.resource("s3").Bucket(bucket).download_file(model_key, "model.tar.gz")
os.system("tar -zxvf model.tar.gz")
```

In [ ]:
```
Kmeans_model_params = mx.ndarray.load("model_algo-1")
```