what-if-tool-challenge

September 29, 2021

1 What-If Tool Challenge Lab

In this notebook, you will use mortgage data from NY in 2017 to create two binary classifiers to determine if a mortgage applicant will be granted a loan.

You will train classifiers on two datasets. One will be trained on the complete dataset, and the other will be trained on a subset of the dataset, where 90% of the female applicants that were granted a loan were removed from the training data (so the dataset has 90% less females that were granted loans).

You will then compare and examine the two models using the What-If Tool.

In this notebook, you will be exepcted to: * Understand how the data is processed * Write TensorFlow code to build and train two models * Write code to deploy the the models to AI Platform * Examine the models in the What-If Tool

2 Download and import the data

Here, you'll import some modules and download some data from the Consumer Finance public datasets.

[1]:
```
import pandas as pd
import numpy as np
import collections
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.utils import shuffle
from witwidget.notebook.visualization import WitWidget, WitConfigBuilder
```

[2]:
```
!wget https://files.consumerfinance.gov/hmda-historic-loan-data/
↪hmda_2017_ny_all-records_labels.zip
!unzip hmda_2017_ny_all-records_labels.zip
```

```
--2021-09-29 05:47:18--  https://files.consumerfinance.gov/hmda-historic-loan-
data/hmda_2017_ny_all-records_labels.zip
Resolving files.consumerfinance.gov (files.consumerfinance.gov)… 52.84.162.85,
52.84.162.24, 52.84.162.10, …
Connecting to files.consumerfinance.gov
```

```
(files.consumerfinance.gov)|52.84.162.85|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 17466285 (17M) [application/zip]
Saving to: 'hmda_2017_ny_all-records_labels.zip'

hmda_2017_ny_all-re 100%[===================>]  16.66M  --.-KB/s    in 0.1s

2021-09-29 05:47:18 (141 MB/s) - 'hmda_2017_ny_all-records_labels.zip' saved
[17466285/17466285]

Archive:  hmda_2017_ny_all-records_labels.zip
  inflating: hmda_2017_ny_all-records_labels.csv
```

3 Process the Data

In this section, you **don't need to write any code**. We suggest you read through the cells to understand how the dataset is processed.

Here, we start by importing the dataset into a Pandas dataframe. Then we process the data to exclude incomplete information and make a simple binary classification of loan approvals. We then create two datasets, one complete and one where 90% of female applicants are removed.

[3]:
```python
# Set column dtypes for Pandas
column_names = collections.OrderedDict({
  'as_of_year': np.int16,
  'agency_abbr': 'category',
  'loan_type': 'category',
  'property_type': 'category',
  'loan_purpose': 'category',
  'owner_occupancy': np.int8,
  'loan_amt_000s': np.float64,
  'preapproval': 'category',
  'county_code': np.float64,
  'applicant_income_00s': np.float64,
  'purchaser_type': 'category',
  'hoepa_status': 'category',
  'lien_status': 'category',
  'population': np.float64,
  'ffiec_median_fam_income': np.float64,
  'tract_to_msamd_income': np.float64,
  'num_of_owner_occupied_units': np.float64,
  'number_of_1_to_4_family_units': np.float64,
  'approved': np.int8,
  'applicant_race_name_3': 'category',
  'applicant_race_name_4': 'category',
  'applicant_race_name_5': 'category',
  'co_applicant_race_name_3': 'category',
  'co_applicant_race_name_4': 'category',
```

```
    'co_applicant_race_name_5': 'category'
})

# Import the CSV into a dataframe
data = pd.read_csv('hmda_2017_ny_all-records_labels.csv', dtype=column_names)
data = shuffle(data, random_state=2)
```

3.1 Extract columns and create dummy dataframes

We first specify which columns to keep then drop the columns that don't have `loan originated` or `loan denied`, to make this a simple binary classification.

We then create two dataframes `binary_df` and `bad_binary_df`. The first will include all the data, and the second will have 90% of female applicants removed, respectively. We then convert them into "dummy" dataframes to turn categorical string features into simple 0/1 features and normalize all the columns.

[4]:
```
# Only use a subset of the columns for these models
text_columns_to_keep = [
            'agency_name',
            'loan_type_name',
            'property_type_name',
            'loan_purpose_name',
            'owner_occupancy_name',
            'applicant_ethnicity_name',
            'applicant_race_name_1',
            'applicant_sex_name',
]
numeric_columns_to_keep = [
            'loan_amount_000s',
            'applicant_income_000s',
            'population',
            'minority_population',
            'hud_median_family_income'
]

columns_to_keep = text_columns_to_keep + numeric_columns_to_keep +␣
 ↪['action_taken_name']

# Drop columns with incomplete information and drop columns that don't have␣
 ↪loan orignated or denied, to make this a simple binary classification
df = data[columns_to_keep].dropna()
binary_df = df[df.action_taken_name.isin(['Loan originated', 'Application␣
 ↪denied by financial institution'])].copy()
binary_df.loc[:,'loan_granted'] = np.where(binary_df['action_taken_name'] ==␣
 ↪'Loan originated', 1, 0)
binary_df = binary_df.drop(columns=['action_taken_name'])
```

```
# Drop 90% of loaned female applicants for a "bad training data" version
loaned_females = (binary_df['applicant_sex_name'] == 'Female') &␣
↪(binary_df['loan_granted'] == 1)
bad_binary_df = binary_df.drop(binary_df[loaned_females].sample(frac=.9).index)
```

[5]:
```
# Now lets' see the distribution of approved / denied classes (0: denied, 1:␣
↪approved)
print(binary_df['loan_granted'].value_counts())
```

```
1    223026
0     63001
Name: loan_granted, dtype: int64
```

[6]:
```
# Turn categorical string features into simple 0/1 features (like turning "sex"␣
↪into "sex_male" and "sex_female")
dummies_df = pd.get_dummies(binary_df, columns=text_columns_to_keep)
dummies_df = dummies_df.sample(frac=1).reset_index(drop=True)

bad_dummies_df = pd.get_dummies(bad_binary_df, columns=text_columns_to_keep)
bad_dummies_df = bad_dummies_df.sample(frac=1).reset_index(drop=True)
```

[7]:
```
# Normalize the numeric columns so that they all have the same scale to␣
↪simplify modeling/training
def normalize():
  min_max_scaler = preprocessing.MinMaxScaler()
  column_names_to_normalize = ['loan_amount_000s', 'applicant_income_000s',␣
↪'minority_population', 'hud_median_family_income', 'population']
  x = dummies_df[column_names_to_normalize].values
  x_scaled = min_max_scaler.fit_transform(x)
  df_temp = pd.DataFrame(x_scaled, columns=column_names_to_normalize, index =␣
↪dummies_df.index)
  dummies_df[column_names_to_normalize] = df_temp

  x = bad_dummies_df[column_names_to_normalize].values
  x_scaled = min_max_scaler.fit_transform(x)
  bad_df_temp = pd.DataFrame(x_scaled, columns=column_names_to_normalize, index␣
↪= bad_dummies_df.index)
  bad_dummies_df[column_names_to_normalize] = bad_df_temp

normalize()
```

3.2 Get the Train & Test Data

Now, let's get the train and test data for our models.

For the **first** model, you'll use `train_data` and `train_labels`.

For the **second** model, you'll use `limited_train_data` and `limited_train_labels`.

[8]:
```
# Get the training data & labels
test_data_with_labels = dummies_df

train_data = dummies_df
train_labels = train_data['loan_granted']
train_data = train_data.drop(columns=['loan_granted'])

# Get the bad (limited) training data and labels
limited_train_data = bad_dummies_df
limited_train_labels = limited_train_data['loan_granted']
limited_train_data = bad_dummies_df.drop(columns=['loan_granted'])

# Split the data into train / test sets for Model 1
x,y = train_data,train_labels
train_data,test_data,train_labels,test_labels = train_test_split(x,y)

# Split the bad data into train / test sets for Model 2
lim_x,lim_y=limited_train_data,limited_train_labels
limited_train_data,limited_test_data,limited_train_labels,limited_test_labels =␣
↪train_test_split(lim_x,lim_y)
```

4 Create and train your TensorFlow models

In this section, you will write code to train two TensorFlow Keras models.

4.1 Train your first model on the complete dataset.

- **Important**: your first model should be saved in the location `saved_complete_model/saved_model.pb`.
- The data will come from `train_data` and `train_labels`.

If you get stuck, you can view the documentation here.

[9]:
```
# import TF modules
from tensorflow.keras import layers
from tensorflow.keras import initializers
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

[10]:
```
# This is the size of the array you'll be feeding into our model for each␣
↪example
input_size = len(train_data.iloc[0])

# Train the first model on the complete dataset. Use `train_data` for your data␣
↪and `train_labels` for you labels.
```

```python
# ---- TODO ---------
# create the model = Sequential()
# model.add (your layers)
# model.compile
# model.fit
model = Sequential()
model.add(layers.Dense(200, input_shape=(input_size,), activation='relu'))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])
model.fit(train_data, train_labels, epochs=10, batch_size=2048,␣
 ↪validation_split=0.1)
```

```
2021-09-29 05:47:55.300665: I
tensorflow/core/platform/profile_utils/cpu_utils.cc:104] CPU Frequency:
2200205000 Hz
2021-09-29 05:47:55.301586: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x5574e8e6f490 initialized for platform Host (this does not
guarantee that XLA will be used). Devices:
2021-09-29 05:47:55.301628: I tensorflow/compiler/xla/service/service.cc:176]
StreamExecutor device (0): Host, Default Version
2021-09-29 05:47:55.332596: I
tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool
with default inter op setting: 2. Tune using inter_op_parallelism_threads for
best performance.
```

```
Epoch 1/10
95/95 [==============================] - 4s 37ms/step - loss: 0.1601 - accuracy:
0.7819 - val_loss: 0.1521 - val_accuracy: 0.7893
Epoch 2/10
95/95 [==============================] - 2s 17ms/step - loss: 0.1512 - accuracy:
0.7900 - val_loss: 0.1504 - val_accuracy: 0.7926
Epoch 3/10
95/95 [==============================] - 2s 19ms/step - loss: 0.1504 - accuracy:
0.7923 - val_loss: 0.1511 - val_accuracy: 0.7930
Epoch 4/10
95/95 [==============================] - 3s 26ms/step - loss: 0.1500 - accuracy:
0.7930 - val_loss: 0.1499 - val_accuracy: 0.7935
Epoch 5/10
95/95 [==============================] - 3s 34ms/step - loss: 0.1496 - accuracy:
0.7937 - val_loss: 0.1497 - val_accuracy: 0.7931
Epoch 6/10
95/95 [==============================] - 3s 36ms/step - loss: 0.1495 - accuracy:
0.7934 - val_loss: 0.1498 - val_accuracy: 0.7936
Epoch 7/10
95/95 [==============================] - 3s 33ms/step - loss: 0.1493 - accuracy:
```

```
0.7941 - val_loss: 0.1496 - val_accuracy: 0.7930
Epoch 8/10
95/95 [==============================] - 3s 32ms/step - loss: 0.1492 - accuracy:
0.7943 - val_loss: 0.1499 - val_accuracy: 0.7934
Epoch 9/10
95/95 [==============================] - 2s 26ms/step - loss: 0.1489 - accuracy:
0.7942 - val_loss: 0.1494 - val_accuracy: 0.7930
Epoch 10/10
95/95 [==============================] - 2s 18ms/step - loss: 0.1489 - accuracy:
0.7944 - val_loss: 0.1496 - val_accuracy: 0.7939
```

[10]:
```
<tensorflow.python.keras.callbacks.History at 0x7f0a03553e50>
```

[11]:
```
# Save your model
!mkdir -p saved_complete_model
model.save('saved_complete_model')
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-
packages/tensorflow/python/training/tracking/tracking.py:111:
Model.state_updates (from tensorflow.python.keras.engine.training) is deprecated
and will be removed in a future version.
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied
automatically.
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-
packages/tensorflow/python/training/tracking/tracking.py:111: Layer.updates
(from tensorflow.python.keras.engine.base_layer) is deprecated and will be
removed in a future version.
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied
automatically.

2021-09-29 05:48:26.020183: W tensorflow/python/util/util.cc:348] Sets are not
currently considered sequences, but this may change in the future, so consider
avoiding using them.

INFO:tensorflow:Assets written to: saved_complete_model/assets
```

[12]:
```
# Get predictions on the test set and print the accuracy score (Model 1)
y_pred = model.predict(test_data)
acc = accuracy_score(test_labels, y_pred.round())
print("Model 1 Accuracy: %.2f%%" % (acc * 100.0))
```

```
Model 1 Accuracy: 79.05%
```

4.2 Train your second model on the limited datset.

- **Important**: your second model should be saved in the location `saved_limited_model/saved_model.pb`.

- The data will come from `limited_train_data` and `limited_train_labels`.

If you get stuck, you can view the documentation here.

[13]:
```
# Train your second model on the limited dataset. Use `limited_train_data` for␣
↪your data and `limited_train_labels` for your labels.
# Use the same input_size for the limited_model

# ---- TODO ---------
# create the limited_model = Sequential()
# limited_model.add (your layers)
# limited_model.compile
# limited_model.fit
limited_model = Sequential()
limited_model.add(layers.Dense(200, input_shape=(input_size,),␣
↪activation='relu'))
limited_model.add(layers.Dense(50, activation='relu'))
limited_model.add(layers.Dense(20, activation='relu'))
limited_model.add(layers.Dense(1, activation='sigmoid'))
limited_model.compile(loss='mean_squared_error', optimizer='adam',␣
↪metrics=['accuracy'])
limited_model.fit(limited_train_data, limited_train_labels, epochs=10,␣
↪batch_size=2048, validation_split=0.1)
```

```
Epoch 1/10
74/74 [==============================] - 3s 39ms/step - loss: 0.1759 - accuracy:
0.7456 - val_loss: 0.1540 - val_accuracy: 0.7851
Epoch 2/10
74/74 [==============================] - 1s 16ms/step - loss: 0.1532 - accuracy:
0.7863 - val_loss: 0.1523 - val_accuracy: 0.7889
Epoch 3/10
74/74 [==============================] - 1s 15ms/step - loss: 0.1522 - accuracy:
0.7883 - val_loss: 0.1525 - val_accuracy: 0.7889
Epoch 4/10
74/74 [==============================] - 2s 27ms/step - loss: 0.1518 - accuracy:
0.7890 - val_loss: 0.1518 - val_accuracy: 0.7904
Epoch 5/10
74/74 [==============================] - 2s 31ms/step - loss: 0.1514 - accuracy:
0.7899 - val_loss: 0.1516 - val_accuracy: 0.7905
Epoch 6/10
74/74 [==============================] - 2s 30ms/step - loss: 0.1514 - accuracy:
0.7905 - val_loss: 0.1517 - val_accuracy: 0.7895
Epoch 7/10
74/74 [==============================] - 2s 24ms/step - loss: 0.1512 - accuracy:
0.7900 - val_loss: 0.1514 - val_accuracy: 0.7904
Epoch 8/10
74/74 [==============================] - 3s 35ms/step - loss: 0.1509 - accuracy:
0.7905 - val_loss: 0.1516 - val_accuracy: 0.7908
```

```
Epoch 9/10
74/74 [==============================] - 3s 41ms/step - loss: 0.1508 - accuracy:
0.7904 - val_loss: 0.1513 - val_accuracy: 0.7899
Epoch 10/10
74/74 [==============================] - 3s 35ms/step - loss: 0.1506 - accuracy:
0.7909 - val_loss: 0.1514 - val_accuracy: 0.7913
```

[13]:
```
<tensorflow.python.keras.callbacks.History at 0x7f09cc445a10>
```

[14]:
```
# Save your model
!mkdir -p saved_limited_model
limited_model.save('saved_limited_model')
```

```
INFO:tensorflow:Assets written to: saved_limited_model/assets
```

[15]:
```
# Get predictions on the test set and print the accuracy score (Model 2)
limited_y_pred = limited_model.predict(limited_test_data)
acc = accuracy_score(limited_test_labels, limited_y_pred.round())
print("Model 2 Accuracy: %.2f%%" % (acc * 100.0))
```

```
Model 2 Accuracy: 78.81%
```

5 Deploy your models to the AI Platform

In this section, you will first need to create a Cloud Storage bucket to store your models, then you will use gcloud commands to copy them over.

You will then create two AI Platform model resources and their associated versions.

[45]:
```
# ---- TODO ---------

# Fill out this information:

GCP_PROJECT = 'qwiklabs-gcp-01-e13fbd3509d2'
MODEL_BUCKET = 'gs://qwiklabs-gcp-01-e13fbd3509d2'
MODEL_NAME = 'complete_model' #do not modify
LIM_MODEL_NAME = 'limited_model' #do not modify
VERSION_NAME = 'v1'
REGION = 'us-central1'
```

[46]:
```
# Copy your model files to Cloud Storage (these file paths are your 'origin'␣
↪for the AI Platform Model)
!gsutil cp -r ./saved_complete_model $MODEL_BUCKET
!gsutil cp -r ./saved_limited_model $MODEL_BUCKET
```

```
Copying file://./saved_complete_model/saved_model.pb [Content-
Type=application/octet-stream]…
Copying file://./saved_complete_model/variables/variables.index [Content-
Type=application/octet-stream]…
```

```
Copying file://./saved_complete_model/variables/variables.data-00000-of-00001
[Content-Type=application/octet-stream]…
- [3 files][332.2 KiB/332.2 KiB]
Operation completed over 3 objects/332.2 KiB.
Copying file://./saved_limited_model/saved_model.pb [Content-
Type=application/octet-stream]…
Copying file://./saved_limited_model/variables/variables.index [Content-
Type=application/octet-stream]…
Copying file://./saved_limited_model/variables/variables.data-00000-of-00001
[Content-Type=application/octet-stream]…
- [3 files][332.9 KiB/332.9 KiB]
Operation completed over 3 objects/332.9 KiB.
```

[47]:

```
# Configure gcloud to use your project
!gcloud config set project $GCP_PROJECT
!gcloud config set ai_platform/region global
```

```
Updated property [core/project].
Updated property [ai_platform/region].
```

5.1 Create your first AI Platform model: *saved_complete_model*

Navigate back to the Google Cloud Console to complete this task. See the lab guide for details.

[48]:

```
!gcloud ai-platform models create $MODEL_NAME --regions $REGION
```

```
Using endpoint [https://ml.googleapis.com/]
Created ai platform model [projects/qwiklabs-
gcp-01-e13fbd3509d2/models/complete_model].
```

[51]:

```
!gcloud ai-platform versions create $VERSION_NAME \
--model=$MODEL_NAME \
--framework='TENSORFLOW' \
--runtime-version=2.3\
--origin=$MODEL_BUCKET/saved_complete_model\
--staging-bucket=$MODEL_BUCKET \
--python-version=3.7
```

```
Using endpoint [https://ml.googleapis.com/]
Creating version (this might take a few minutes)…done.
```

5.2 Create your second AI Platform model: *saved_limited_model*

Navigate back to the Google Cloud Console to complete this task. See the lab guide for details.

[52]:

```
!gcloud ai-platform models create $LIM_MODEL_NAME --regions $REGION
```

```
Using endpoint [https://ml.googleapis.com/]
Created ai platform model [projects/qwiklabs-
```

gcp-01-e13fbd3509d2/models/limited_model].

[53]:
```
# ---- TODO ------
!gcloud ai-platform versions create $VERSION_NAME \
--model=$LIM_MODEL_NAME \
--framework='TENSORFLOW' \
--runtime-version=2.3 \
--origin=$MODEL_BUCKET/saved_limited_model\
--staging-bucket=$MODEL_BUCKET \
--python-version=3.7
```

```
Using endpoint [https://ml.googleapis.com/]
Creating version (this might take a few minutes)…done.
```

6 Using the What-if Tool to interpret your model

Once your models have deployed, you're now ready to connect them to the What-if Tool using the WitWidget.

We've provided the Config Builder code and a couple of functions to get the class predictions from the models, which are necessary inputs for the WIT. If you've successfully deployed and saved your models, all you'll need to do is **add the WitConfigBuilder code in this cell**.

[55]:
```
#@title Show model results in WIT
num_datapoints = 1000  #@param {type: "number"}

# Column indices to strip out from data from WIT before passing it to the model.
columns_not_for_model_input = [
    test_data_with_labels.columns.get_loc('loan_granted'),
]

# Return model predictions.
def custom_predict(examples_to_infer):
  # Delete columns not used by model
  model_inputs = np.delete(
      np.array(examples_to_infer), columns_not_for_model_input, axis=1).tolist()
  # Get the class predictions from the model.
  preds = model.predict(model_inputs)
  preds = [[1 - pred[0], pred[0]] for pred in preds]
  return preds

# Return 'limited' model predictions.
def bad_custom_predict(examples_to_infer):
  # Delete columns not used by model
  model_inputs = np.delete(
      np.array(examples_to_infer), columns_not_for_model_input, axis=1).tolist()
  # Get the class predictions from the model.
  preds = limited_model.predict(model_inputs)
```

```
  preds = [[1 - pred[0], pred[0]] for pred in preds]
  return preds

examples_for_wit = test_data_with_labels.values.tolist()
column_names = test_data_with_labels.columns.tolist()

# ---- TODO ------

## Add WitConfigBuilder code here
```

[ ]:

```

```