

Using Machine Learning to Detect and Predict the Likelihood of Heart Attack

Introduction

Heart diseases or Cardiovascular diseases (CVDs) are the number one cause of death globally, claiming an estimated 17.9 million lives yearly. The World Health Organization estimates this to be 31% of deaths worldwide. In the United States alone, the Center for Disease Control estimates that a heart attack occurs every forty seconds translating to 805,000 Americans each year.

Some common symptoms include pain and tightness in the chest, shortness of breath, cold sweat, fatigue, sudden dizziness among others. Not everybody who has a heart attack experiences the same type or severity of symptoms. People experience moderate to intense pain while some have no such symptoms. It is expected that the higher the number of symptoms experienced the greater the likelihood of heart disease.

The severity and prevalence of this disease have necessitated a massive need for AI to develop predictive models that can help in disease management and risk control.

Our goal is to build ML models for Heart Disease Detection; export the best performing model to Android devices for everyday use. In addition, we will highlight the ML Operations for this project using Katib for Hyperparameter Tuning of the Model and creating a Pipeline using Kubeflow.

Data

The dataset used for this project was taken from Kaggle. It has observations of 303 patients with 14 features such as demographic features like age, gender, fasting blood sugar, cholesterol level, resting blood pressure, and so on. The chances of a heart attack were binarily classified. This classification was taken as the target in our modeling. The Data Dictionary for the dataset is as shown below.

Figure 1: Data Dictionary

- Age: Age of the patient
- Sex: Sex of the patient sex (1 = male; 0 = female)
- exang: exercise induced angina (1 = yes; 0 = no)
- ca: number of major vessels (0-3)
- cp : Chest Pain type
 - Value 0: no pain
 - Value 1: typical angina
 - Value 2: atypical angina
 - Value 3: non-anginal pain
 - Value 4: asymptomatic
- trtbps: resting blood pressure (in mmHg)
- chol: cholesterol in mg/dl fetched via BMI sensor
- fbs: (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
- rest_ecg : resting electrocardiographic results
 - Value 0: normal
 - Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)
 - Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria
- thalachh: maximum heart rate achieved
- oldpeak = ST depression induced by exercise relative to rest
- slp = the slope of the peak exercise ST segment
- target: 0= less chance of heart attack 1= more chance of heart attack

Analysis

A review of the features revealed an imbalance for some categories primarily gender and the output. We noted that the average age is 54.37 years and the outlier observations were in three features namely Cholesterol, Resting Blood Pressure, and Maximum Heart Rate. Chest pain showed the highest correlation with the target.

Figure 2: Proportions of Gender and the Target Label

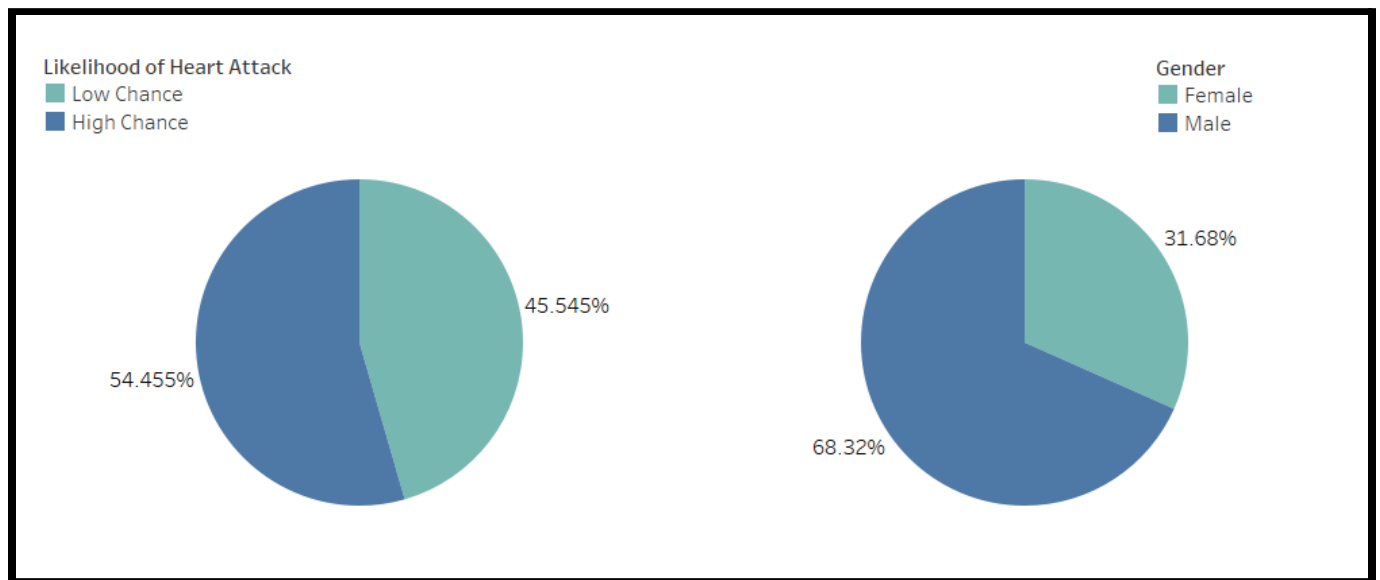


Figure 3: Boxplot showing the Outliers

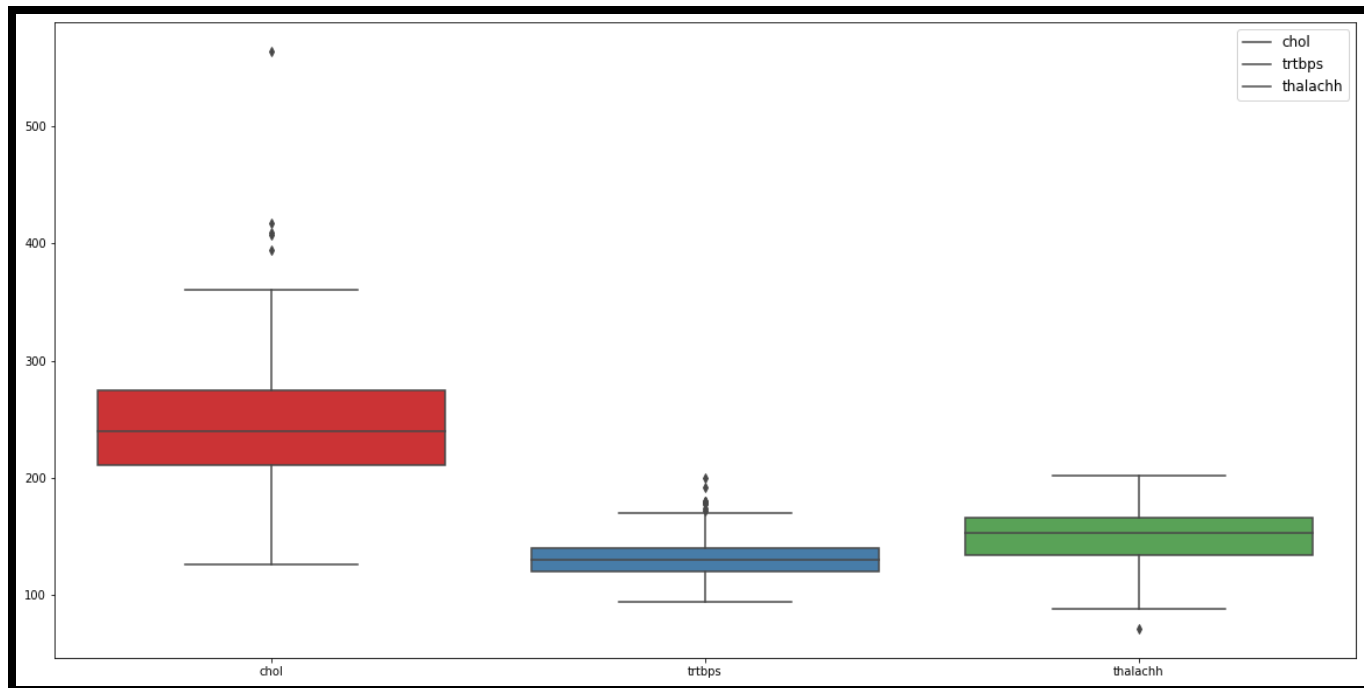
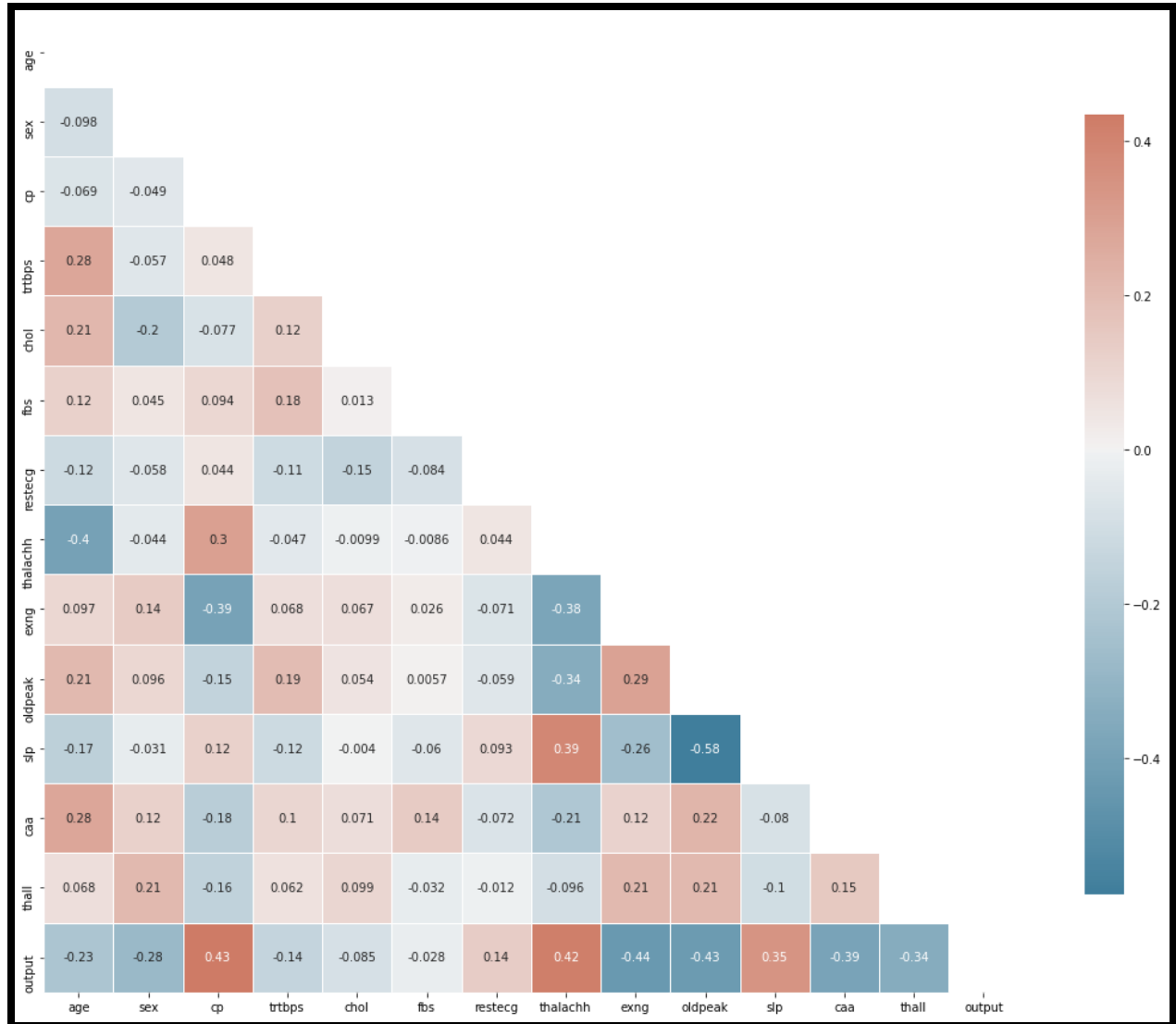


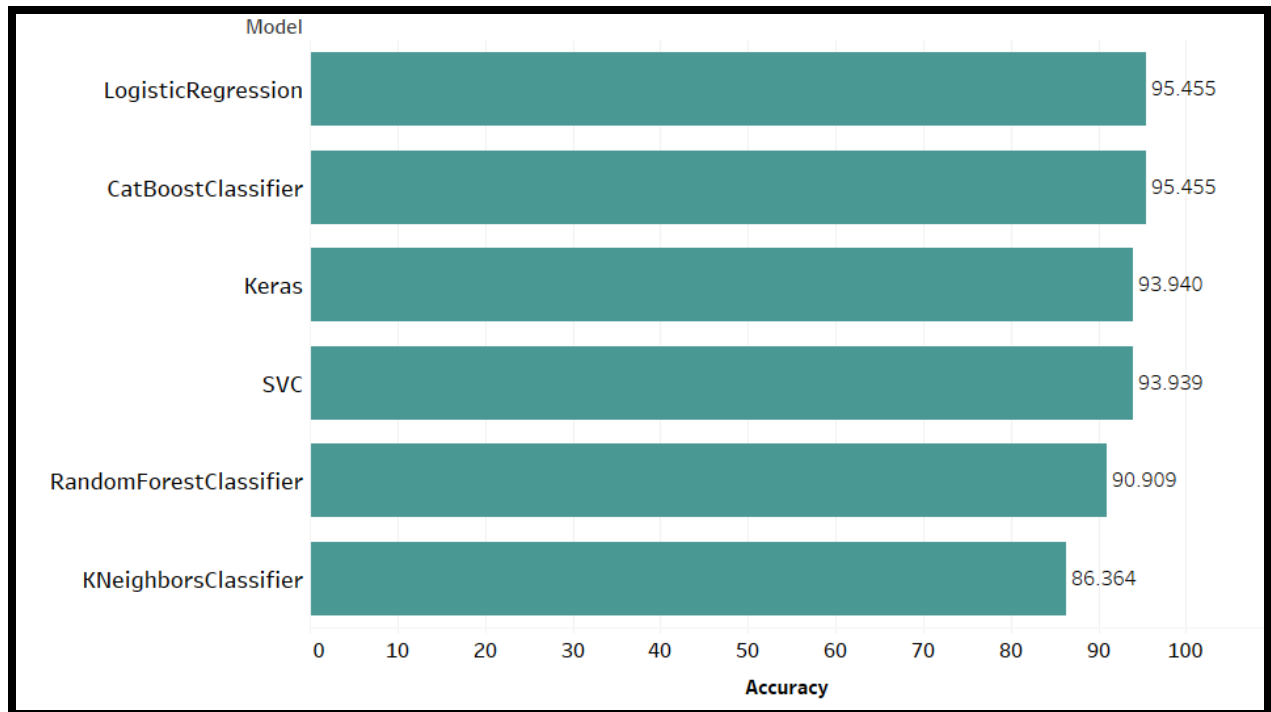
Figure 4: Correlation plot of all the features



Modeling

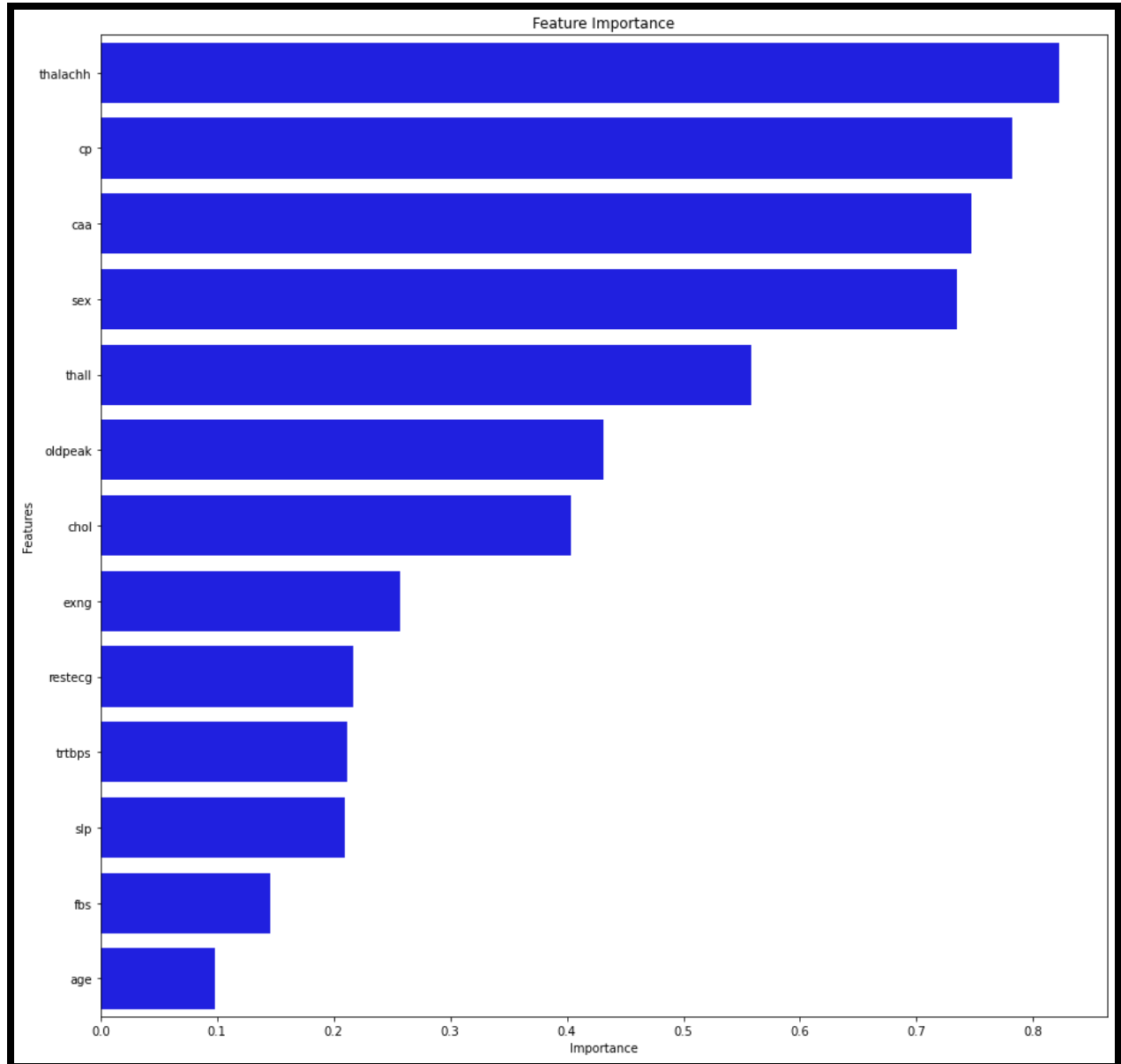
Data preprocessing tasks include removal of outliers, resampling the target variable for a more balanced distribution, and scaling. After preprocessing, our clean data was tested with six models, our best performers were the Logistic Regression and CatBoost models with a tied accuracy of 95%.

Figure 5: Model Accuracy Scores



The Logistic Regression Model showed that the Maximum Heart Rate Achieved (thalachh) feature had the highest influence on the target. The visualization in figure 6 below highlights the Feature Importance Results.

Figure 6: Logistic Regression Feature Importance



Converting Machine Learning Model to Tensorflow Lite for Android devices

The Keras Model was converted into a TensorflowLite format to be used on Android devices. The TensorFlow Lite Converter did not support Logistic Regression or CatBoost Classifier Models at the time the project was conducted.

The conversion steps are as follows:

1. Import TensorFlow Lite:

```
pip install tfLite
```

2. Convert the Keras model to TensorFlow lite format:

```
converter = tf.lite.TFLiteConverter.from_keras_model(keras_model)
tflite_model = converter.convert()
```

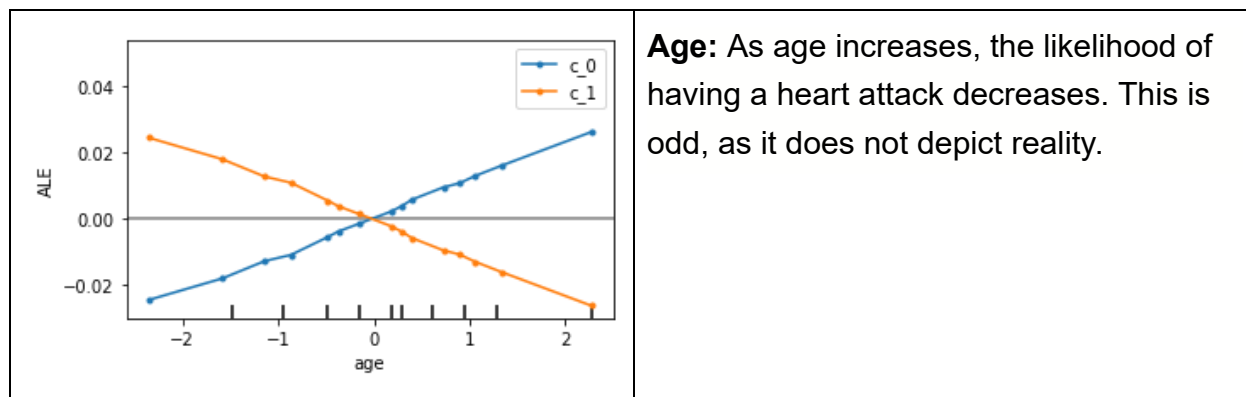
3. Save the model:

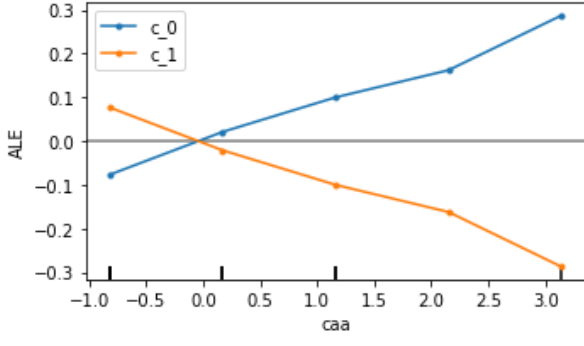
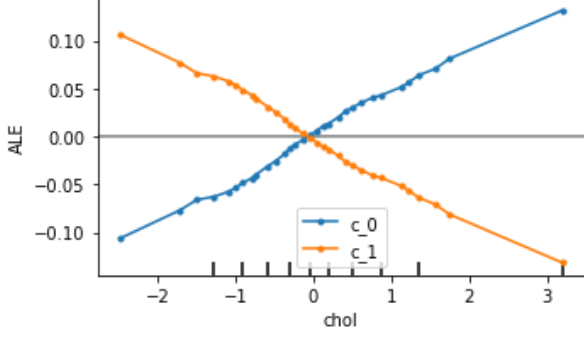
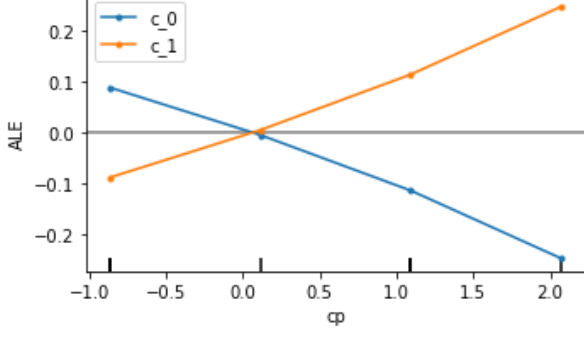
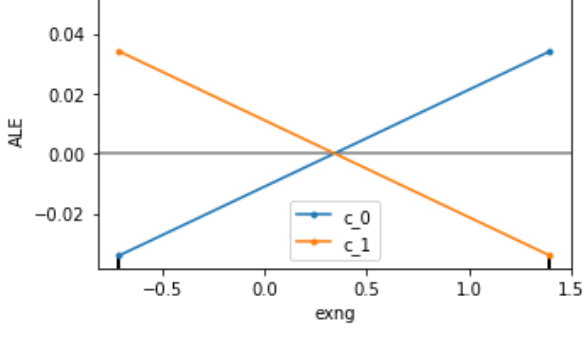
```
with open('HeartFailurePrediction_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

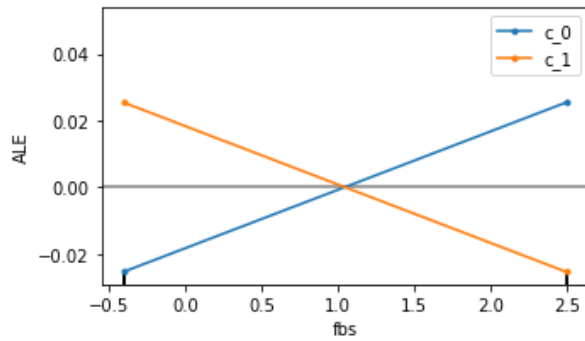
Model Explainability with Alibi Explain's ALE (Accumulated Local Effects) Plots

In this session, we will use the [ALE](#) explainer (Accumulated Local Effects) plots to explain the behavior of our best performing model, the Logistic Regression model on our dataset.

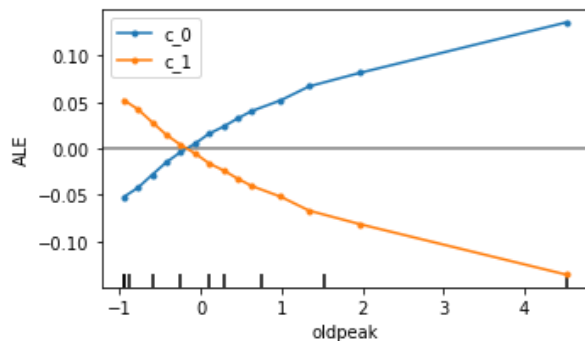
Figure 7: ALE on Logistic Regression Model



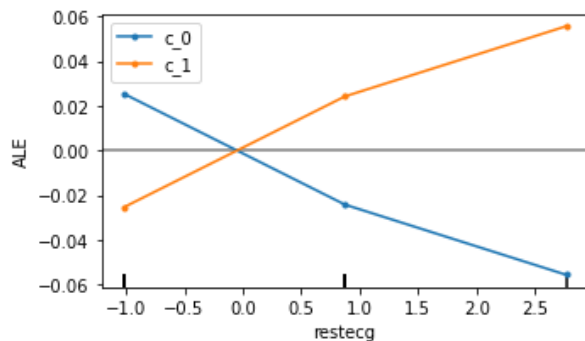
	<p>caa (number of major vessels (0-3) colored by fluoroscopy): For $caa = 0$, there is no effect on average predictions, but as caa increases, its effect on classifying the instance as 1, decreases.</p>
	<p>chol: For higher cholesterol levels, the model assigns negative probabilities towards classifying instances as 1 while for lower cholesterol levels, the model assigns higher probabilities towards classifying the instance as 0.</p>
	<p>cp: The model assigns positive probabilities towards classifying chest pain types 1 and 2 as 1 and negative probabilities towards classifying them as 0. Chest pain type 0 seems to not affect average prediction.</p>
	<p>exng: For instances where $exng = 0$ (that is, patients with no exercise-induced angina), there is no effect on average prediction. For instances where $exng = 1$ (that is, patients with exercise-induced angina), their effects on average prediction are negative.</p>



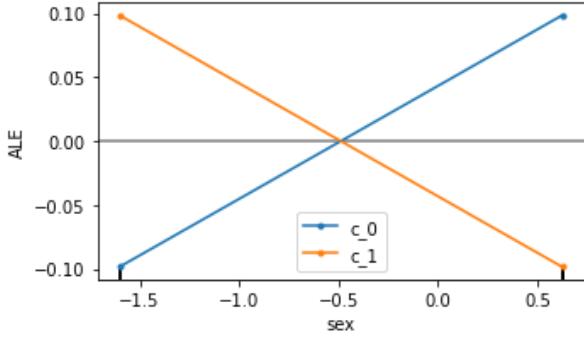
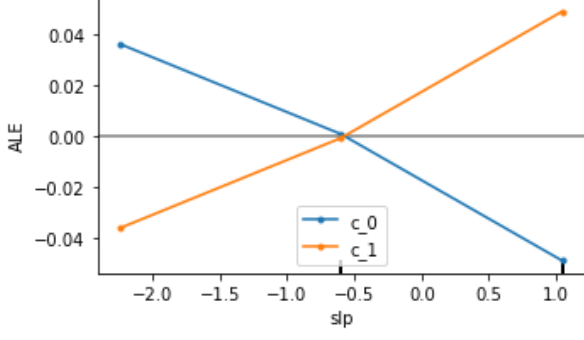
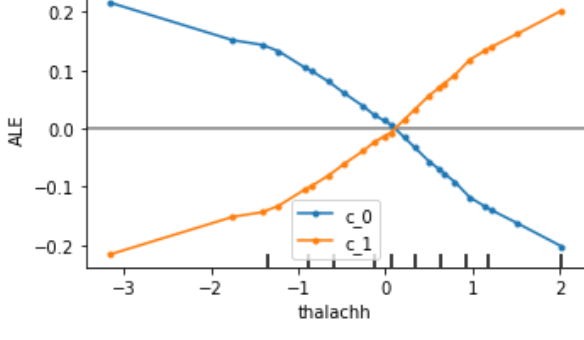
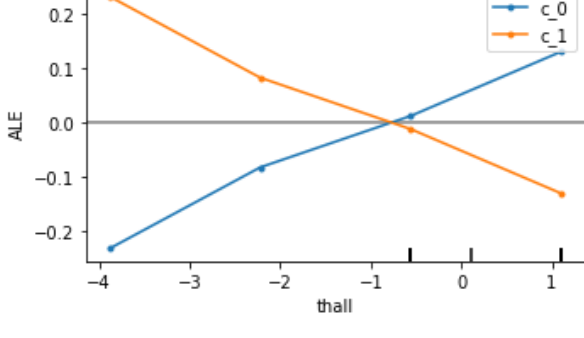
fbs: For instances where fbs =1 (that is fasting blood sugar > 120), there seems to be no effect on average prediction, but for instances where fbs=0 (that is fasting blood sugar < 120), the model seems to assign positive probabilities towards classifying those instances as 1.

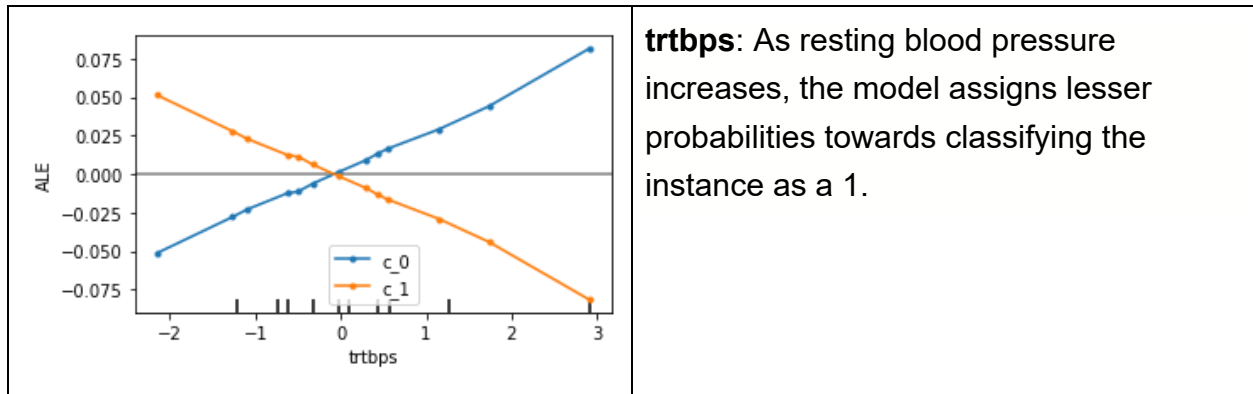


oldpeak (ST depression induced by exercise relative to rest): As old peak increases, its effect on classifying the instance as a 1 decreases.



rest_ecg: For instances where restecg (resting electrocardiographic results) = 0, (that is normal restecgs) results, there's no effect on average predictions. For instances where rest_ecgs = 1 (that is having ST-T wave abnormality), the model assigns positive probabilities towards classifying those instances as 1. For instances where rest_ecg = 2(that is showing probable or definite left ventricular hypertrophy by Estes' criteria), the model assigns even higher positive probabilities towards classifying those instances as 1.

	<p>Sex: At sex =0 and 1, the model assigns negative probabilities towards classifying the instance as 1 and positive probabilities towards classifying them as 0. So gender does not seem to have much effect on average prediction.</p>
	<p>slp: (the slope of the peak exercise ST segment): For slp =1, the model assigns positive probabilities towards classifying the instance as 1.</p>
	<p>thalachh: As the maximum heart rate achieved increases, the model assigns increasingly higher probabilities towards classifying those instances as 1. This means that patients with higher maximum heart rate achieved are more likely to have heart attacks.</p>
	<p>thall: As thall increases, its effect on classifying an instance as 1 decreases.</p>



The ALE plots show maximum heart rate achieved as the feature, most effective in predicting heart failure. According to heart.org, older age and high cholesterol levels are factors that increase the risk of a heart attack. Our dataset, however, does not depict that.

Implementing Kubeflow ML Operators for Model Training

An Operator is a method of packaging, deploying and managing a stateful Kubernetes application which in this context is a Machine Learning Job. [Operators](#) are software written to encapsulate all of those operational considerations for a specific Kubernetes application and ensure that all aspects of its lifecycle, from configuration and deployment to upgrades, monitoring, and failure-handling, are integrated right into the Kubernetes framework and invoked when needed. An ML Operator can be made for a range of actions from basic functionalities to specific logic for an ML Job.

Tensorflow Operator

This is one of the operators offered by [Kubeflow](https://kubeflow.org/) to make it easy to run and monitor both distributed and non-distributed tensorflow jobs on Kubernetes. Training tensorflow models using tf-operator relies on centralized parameter servers for coordination between workers. It supports the tensorflow framework only.

Tensorflow Training Jobs (TFJob)

TensorFlow Training Job (TFJob) is a Kubernetes custom resource with a [YAML](#) representation that you can use to run TensorFlow training tasks on Kubernetes. The Kubeflow implementation of TFJob is in [tf-operator](#).

Tensorflow Operator for Heart Attack Dataset

Here, we go through the process of creating a TensorFlow Operator with our Dataset:

1. Check that the right image, [TensorFlow](#) is available:

```
#!/ pip3 list | grep tensorflow
! pip3 install --user tensorflow==2.4.0
! pip3 install --user ipywidgets nbconvert
!python -m pip install --user --upgrade pip
!pip3 install pandas scikit-learn keras
tensorflow-datasets --user
```

2. To package the trainer in a container image, we shall need a file (on our cluster) that contains the code as well as a file with the resource definition of the job for the Kubernetes cluster:

```
TRAINER_FILE = "tfjobheart.py"
KUBERNETES_FILE = "tfjob-heartdisease.yaml"
```

3. Define a helper function to capture output from a cell with %%capture that looks like some-resource created:

```
import re

from IPython.utils.capture import CapturedIO

def get_resource(captured_io: CapturedIO) -> str:
    """
```

```
Gets a resource name from `kubectl apply -f
<configuration.yaml>`.
```

```
:param str captured_io: Output captured by using `%%capture` cell
magic
:return: Name of the Kubernetes resource
:rtype: str
:raises Exception: if the resource could not be created
"""

out = captured_io.stdout
matches = re.search(r"^(.+)\s+created", out)
if matches is not None:
    return matches.group(1)
else:
    raise Exception(f"Cannot get resource as its creation failed:
{out}. It may already exist.")
```

4. Load and Inspect the Data:

```
import pandas as pd
data = pd.read_csv("heart.csv")
data.head()
```

5. Train the Model in the Notebook:

We trained the model in a distributed fashion and put all the code in a single cell. That way we could save the file and include it in a container image. That saves the file as defined by TRAINER_FILE but it does not run it.

```
%%writefile $TRAINER_FILE
import argparse
import logging
import json
import os
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

import numpy as np
import pandas as pd
```

```

from sklearn.model_selection import train_test_split as tts
from sklearn.preprocessing import StandardScaler

from numpy.random import seed

import tensorflow as tf
tf.random.set_seed(221)
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout,
BatchNormalization
from tensorflow.keras.optimizers import SGD, Adam, RMSprop

logging.getLogger().setLevel(logging.INFO)

def make_datasets_unbatched():
    data = pd.read_csv("heart.csv")
    data.head()

    data.apply(lambda x: sum(x.isnull()),axis=0)

    # List of variables with missing values

    vars_with_na=[var for var in data.columns if
data[var].isnull().sum(>1)]

    #Boolean variables
    bool_var=['sex','output','fbs', 'exng']
    #Categorical variables:cardinality
    cat_var=['cp','restecg','slp', 'thall']
    #discrete variables
    num_var=['age', 'trtbps','chol', 'thalachh', 'oldpeak', 'caa']
    #remove outliers

    def removeOutlier(att, data):
        lowerbound = att.mean() - 3 * att.std()
        upperbound = att.mean() + 3 * att.std()
        #print('lowerbound: ',lowerbound,' ----- upperbound: ',
upperbound )
        df1 = data[(att > lowerbound) & (att < upperbound)]
        #print((data.shape[0] - df1.shape[0]), ' number of outliers
from ', data.shape[0] )

```

```

        #print('
*****')
        data = df1.copy()
        return data
    data = removeOutlier(data.trtbps, data)
    data = removeOutlier(data.chol, data)

    #resampling
    from sklearn.utils import resample

    # Separate Target Classes
    df_1 = data[data.output==1]
    df_2 = data[data.output==0]

    # Upsample minority class
    df_upsample_1 = resample(df_2,
                             replace=True,      # sample with
replacement                             n_samples=163,    # to match
majority class                             random_state=123) # reproducible
results
    # Combine majority class with upsampled minority class
    df_upsampled = pd.concat([df_1, df_upsample_1])

    # Display new class counts
    df_upsampled.output.value_counts()

    x = df_upsampled.drop('output', axis = 1)
    y = df_upsampled['output']

    #Split dataset

    x_train,x_test, y_train, y_test = tts(x,y, test_size = 0.2,
random_state = 111)

    #Scaling
    scaler = StandardScaler()
    x_train = scaler.fit_transform(x_train)
    x_test = scaler.fit_transform(x_test)

```



```

    train_dataset = tf.data.Dataset.from_tensor_slices((x_train,
y_train))
    test_dataset = tf.data.Dataset.from_tensor_slices((x_test,
y_test))
    train = train_dataset.cache().shuffle(2000).repeat()
    return train, test_dataset

def model(args):
    seed(1)
    model = Sequential()
    model.add(Dense(10, activation='relu', input_dim=13))
    #model.add(BatchNormalization())
    model.add(Dense(10, activation='relu'))
    #model.add(Dropout(0.2))
    model.add(Dense(1, activation='sigmoid'))

    model.summary()
    opt = args.optimizer
    model.compile(optimizer=opt,
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    tf.keras.backend.set_value(model.optimizer.learning_rate,
args.learning_rate)
    return model

def main(args):
    # MultiWorkerMirroredStrategy creates copies of all variables in
the model's
    # layers on each device across all workers
    strategy =
tf.distribute.experimental.MultiWorkerMirroredStrategy(
communication=tf.distribute.experimental CollectiveCommunication.AUTO
)
    logging.debug(f"num_replicas_in_sync:
{strategy.num_replicas_in_sync}")
    BATCH_SIZE_PER_REPLICA = args.batch_size
    BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync

    # Datasets need to be created after instantiation of
`MultiWorkerMirroredStrategy`
    train_dataset, test_dataset = make_datasets_unbatched()

```

```

train_dataset = train_dataset.batch(batch_size=BATCH_SIZE)
test_dataset = test_dataset.batch(batch_size=BATCH_SIZE)

# See:
https://www.tensorflow.org/api\_docs/python/tf/data/experimental/DistributeOptions
options = tf.data.Options()
options.experimental_distribute.auto_shard_policy = \
    tf.data.experimental.AutoShardPolicy.DATA

train_datasets_sharded = train_dataset.with_options(options)
test_dataset_sharded = test_dataset.with_options(options)

with strategy.scope():
    # Model building/compiling need to be within
    `strategy.scope()`.
    multi_worker_model = model(args)
    # Keras' `model.fit()` trains the model with specified number
of epochs and
    # number of steps per epoch.
    multi_worker_model.fit(train_datasets_sharded,
                           epochs=50,
                           steps_per_epoch=30)

    eval_loss, eval_acc =
multi_worker_model.evaluate(test_dataset_sharded,
                           verbose=0,
                           steps=10)
    # Log metrics for Katib
    logging.info("loss={:.4f}".format(eval_loss))
    logging.info("accuracy={:.4f}".format(eval_acc))

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("--batch_size",
                        type=int,
                        default=32,
                        metavar="N",
                        help="Batch size for training (default: 128)")
    parser.add_argument("--learning_rate",
                        type=float,
                        default=0.1,
                        metavar="N",

```

```

        help='Initial learning rate')
parser.add_argument("--optimizer",
                    type=str,
                    default='adam',
                    metavar="N",
                    help='optimizer')
parsed_args, _ = parser.parse_known_args()
main(parsed_args)

```

6. Create a Docker Image:

The Docker file looks as follows:

```

FROM tensorflow/tensorflow:2.4.0
RUN pip install tensorflow_datasets pandas scikit-learn keras
COPY tfjobheartdisease.py /
ENTRYPOINT ["python", "/tfjobheart.py", "--batch_size", "64",
"--learning_rate", "0.1", "--optimizer", "adam"]

```

7. Check if the code is correct by running it from within the notebook:

```
%run $TRAINER_FILE --optimizer 'adam'
```

8. Create a Distributed TFJob:

For large training jobs, we wish to run our trainer in a distributed model. Once the notebook server cluster can access the Docker image from the registry, we can launch a distributed TF job.

The specification for a distributed TFJob is defined using YAML:

```

%%writefile $KUBERNETES_FILE
apiVersion: "kubeflow.org/v1"
kind: "TFJob"
metadata:
  name: "hrttd"
  namespace: ekemini # your-user-namespace
spec:
  cleanPodPolicy: None
  tfReplicaSpecs:
    Worker:
      replicas: 2
      restartPolicy: OnFailure
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          containers:
            - name: tensorflow
              # modify this property if you would like to use a custom
image
              image: mavencoddevv/tfjob_heart:v.0.1
              command:
                - "python"
                - "/tfjobheart.py"
                - "--batch_size=64"
                - "--learning_rate=0.1"
                - "--optimizer=adam"

```

9. Deploy the distributed training job:

```

%%capture tf_output --no-stderr
! kubectl create -f $KUBERNETES_FILE

```

```

TF_JOB = get_resource(tf_output)

```

10. See the job status:

```

! kubectl describe $TF_JOB

```

11. See the created pods:

```
! kubectl get pods -l job-name=hrttd
```

12. Stream logs from the worker-0 pod to check the training progress:

```
! kubectl logs -f hrttd-worker-0
```

13. Delete the job:

```
! kubectl delete $TF_JOB
```

14. Check to see if the pod is still up and running:

```
#! kubectl -n ekemini logs -f hrttd
```

Hyperparameter Tuning with Katib for Tensorflow Model

Hyperparameter tuning is the process of optimizing a model's hyperparameter values to maximize the predictive quality of the model. [Katib](#) automates the Hyperparameter Tuning process thereby eliminating errors that arise from manual intervention and also saves much-needed resources. Katib is agnostic to ML Frameworks and supports a variety of traditional Hyperparameter Tuning Algorithms. Its concepts are Experiments, Suggestions, Trials, and WorkerJob which are all Custom Resource Definitions integrated on the Kubernetes Engine.

In a nutshell, an Experiment runs several Trials until an objective is reached. Each Trial evaluates Suggestions which are HP values proposed by the tuning process. The WorkerJob evaluates a Trial and calculates its objective value.

This section shows how to create and configure an Experiment for the TensorFlow training job. In terms of Kubernetes, such an experiment is a Custom Resource Definition (CRD) run by the Katib operator.

How to Create Experiments

1. Set up a few basic definitions that can be reused:

```
TF_EXPERIMENT_FILE = "katibheartdisease-tfjob-experiment.yaml"
```

```
import re

from IPython.utils.capture import CapturedIO

def get_resource(captured_io: CapturedIO) -> str:
    """
    Gets a resource name from `kubectl apply -f
    <configuration.yaml>`.

    :param str captured_io: Output captured by using `%%capture` cell
    magic
    :return: Name of the Kubernetes resource
    :rtype: str
    :raises Exception: if the resource could not be created
    """
    out = captured_io.stdout
    matches = re.search(r"^(.+)\s+created", out)
    if matches is not None:
        return matches.group(1)
    else:
        raise Exception(f"Cannot get resource as its creation failed:
        {out}. It may already exist.")
```

2. TensorFlow: Katib TFJob Experiment

The TFJob definition for this example is based on the tensorflow operator notebook shown earlier. For our experiment, we focused on the learning rate, batch-size and optimizer. The following YAML file describes an Experiment object:

```
%%writefile $TF_EXPERIMENT_FILE
apiVersion: "kubeflow.org/v1beta1"
kind: Experiment
metadata:
  namespace: ekemini
  name: heart
spec:
  parallelTrialCount: 3
  maxTrialCount: 12
  maxFailedTrialCount: 3
  objective:
    type: maximize
    goal: 0.8
    objectiveMetricName: accuracy
  algorithm:
    algorithmName: random
  metricsCollectorSpec:
    kind: StdOut
  parameters:
    - name: learning_rate
      parameterType: double
      feasibleSpace:
        min: "0.01"
        max: "0.1"
    - name: batch_size
      parameterType: int
      feasibleSpace:
        min: "50"
        max: "100"
    - name: optimizer
      parameterType: categorical
      feasibleSpace:
        list:
          - rmsprop
          - adam
  trialTemplate:
    primaryContainerName: tensorflow
    trialParameters:
      - name: learningRate
        description: Learning rate for the training model
        reference: learning_rate
      - name: batchSize
```

```

        description: Batch Size
        reference: batch_size
    - name: optimizer
      description: Training model optimizer (sdg, adam)
      reference: optimizer
  trialSpec:
    apiVersion: "kubeflow.org/v1"
    kind: TFJob
    spec:
      tfReplicaSpecs:
        Worker:
          replicas: 1
          restartPolicy: OnFailure
          template:
            metadata:
              annotations:
                sidecar.istio.io/inject: "false"
            spec:
              containers:
                - name: tensorflow
                  image: mavencoddevv/tfjob_heart:v.0.1
                  command:
                    - "python"
                    - "/tfjobheart.py"
                    - "--batch_size=${trialParameters.batchSize}"
                    -
                    - "--learning_rate=${trialParameters.learningRate}"
                    - "--optimizer=${trialParameters.optimizer}"

```

3. Run and Monitor Experiments:

You can either execute these commands on your local machine with `kubectl` or on the notebook server:

```

%%capture kubectl_output --no-stderr
! kubectl apply -f $TF_EXPERIMENT_FILE

```

The cell magic grabs the output of the `kubectl` command and stores it in an object named `kubectl_output`. From there we can use the utility function we defined earlier:


```
EXPERIMENT = get_resource(kubect1_output)
```

4. See experiment status:

```
! kubect1 describe $EXPERIMENT
```

5. Get the list of created experiments:

```
! kubect1 get experiments
```

6. Get the list of created trials:

```
! kubect1 get trials
```

7. After the experiment is completed, use describe to get the best trial results:

```
! kubect1 describe $EXPERIMENT
```

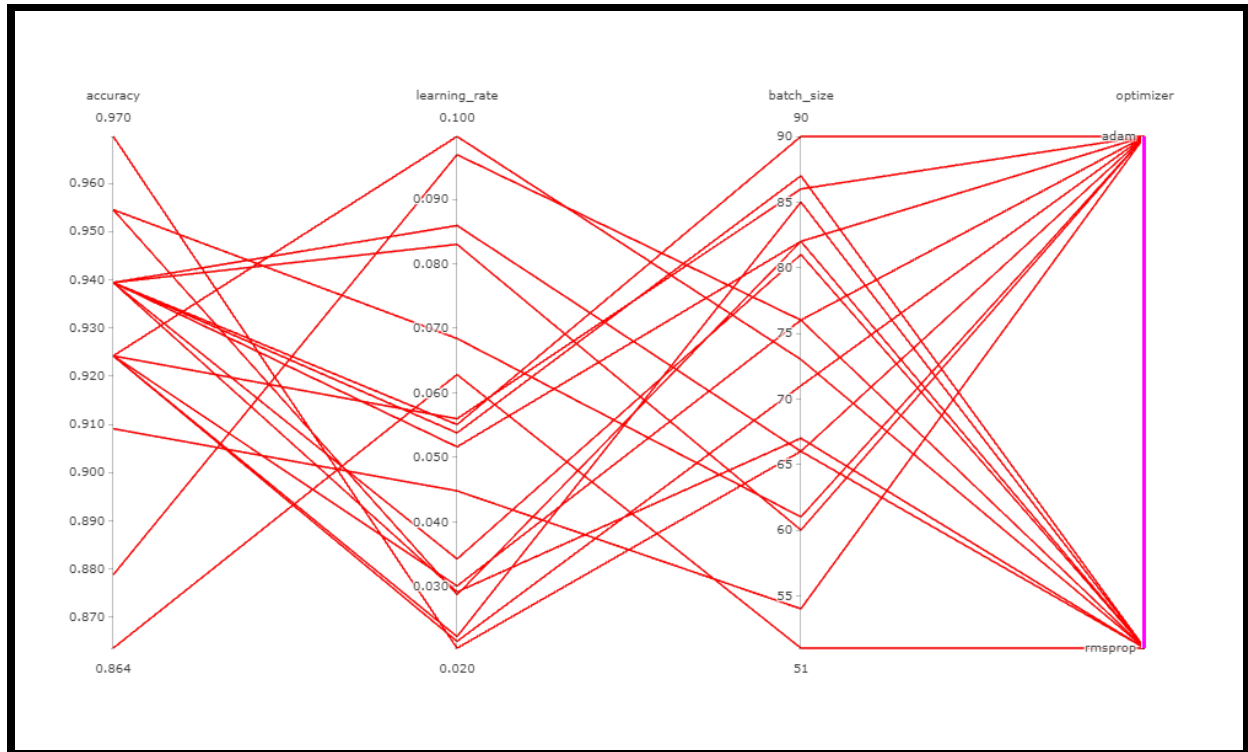
8. Delete Katib job to free up resources:

```
! kubect1 delete -f $TF_EXPERIMENT_FILE
```

9. Check to see if the pod is still up and running:

```
! kubect1 -n ekemini logs -f heart
```

Figure 8: Result of katib experiment



Model Deployment using Kubeflow

Deployment is a crucial factor in the ML process. For the models built to be effective to real-life users, there is a need to position our model on a platform that can successfully receive data from as many users as needed and output the predictions. For this use case, we will make use of the Kubeflow platform which provides helpful services and tools that ease the development, deployment, and management of portable, scalable machine learning projects.

In building Kubeflow Pipelines, the available options to build Kubeflow Pipelines are the Lightweight and Reusable Components. The former is easy to build and update; useful for Testing and Deployment while the latter are stable containerized functions useful for multiple projects. For our use case, the Reusable Components option was adopted.

The process is broken down as follows:

1. Creating self-contained ML code:

When creating reusable components, our first step involves creating functions of our ML code that can pass data between themselves, with all other packages needed to run contained within the function. Each step of the ML process should be packaged in this way.

Figure 9: Logistic Regression Model

```
import argparse
def lr(clean_data):
    import joblib
    import numpy as np
    import pandas as pd
    from sklearn import metrics
    from sklearn.linear_model import LogisticRegression

    data = joblib.load(clean_data)
    X_train = data['X_train']
    y_train = data['Y_train']
    X_test = data['X_test']
    y_test = data['Y_test']

    lr_model = LogisticRegression()

    lr_model.fit(X_train, y_train)

    y_pred = lr_model.predict(X_test)

    # Test score
    test = lr_model.score(X_test, y_test)
    train = lr_model.score(X_train, y_train)
```

```

print('test accuracy:')
print(test)
print('train accuracy:')
print(train)

#Classification Report
report = metrics.classification_report(y_test, y_pred,
output_dict=True)
df_classification_report = pd.DataFrame(report).transpose()
print(df_classification_report)

lr_metrics = {'train':train, 'test':test,
'report':df_classification_report, 'model':lr_model}
joblib.dump(lr_metrics, 'lr_metrics')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--clean_data')
    args = parser.parse_args()
    lr(args.clean_data)

```

2. Create Docker Images:

Using our packaged ML functions, we create Docker Images and push them to the repository where they can be called when needed by the pipeline. By sectioning each step of our code into components, any step can be repeated, scaled, or transformed individually without affecting the other components in the pipeline. Creating Docker Images requires the Docker package in a command line and an account with a repository like DockerHub.

Figure 10: Docker file for our Logistic Regression Component

```
FROM python:3.8
WORKDIR /logistic
RUN pip install -U scikit-learn numpy pandas joblib
COPY logistic.py /logistic
ENTRYPOINT [ "python", "logistic.py" ]
```

This Dockerfile directs the installation of python 3.8 as the base of our image's functioning, installs the needed packages, and creates a working directory for our python function.

d

With both the python function (logistic.py) and the Docker file in the same directory, we can run some Docker Commands to build and push the image to the repository.

Figure 11: Docker Commands

```
# build the image
docker build --tag=lr_heart:v.0.1 .
# tag to a docker repository
docker tag lr_heart:v.0.1 mavencodvv/lr_heart:v.0.1
# push the image to the repository
docker push mavencodvv/lr_heart:v.0.1
```

With mavencodvv as the user id and lr_heart as our image tag, we have successfully built the image for our logistic regression component. Each step of our ML pipeline will be built this way before being compiled using Kubeflow's pipeline functions.

3. Building and Compiling the Pipeline:

Using a Jupyter notebook environment, we will utilize the Kubeflow Python package to build our pipeline from our created images then compile it for deployment.

Figure 12: Kubeflow Software Development Kit

```

# Installing the Kubeflow SDK
!python -m pip install --user --upgrade pip
!pip3 install kfp --upgrade --user

# Restart the runtime then import the packages
import kfp
from kfp import dsl
import kfp.components as comp

```

First, we install the packages then we create component functions built from our created images.

Figure 13: Logistic Regression Component (Kubeflow SDK)

```

def lr_op(clean_data):
    return dsl.ContainerOp(
        name = 'Logistic Regression',
        image = 'mavencodvv/logistic_heart:v.0.1',
        arguments = ['--clean_data', clean_data
                    ],
        file_outputs={
            'lr_metrics': '/logistic/lr_metrics'
        }
    )

```

The inputs and outputs are explicitly stated to facilitate the passage of data between components, once all our pipeline components are packaged in this way we can call them in a final pipeline function that contains all the components created.

Figure 14: Final Pipeline Function

```

@dsl.pipeline(

```

```
    name='Heart Attack Prediction',
    description='An ML reusable pipeline that predicts the chances of
a patient having heart attack'
)

# Define parameters to be fed into pipeline
def heart_pipeline(bucket_name, credentials):

    _load_data_op = load_data_op()

    _stat_op = stat_op(
        dsl.InputArgumentPath(_load_data_op.outputs['data'])
    ).after(_load_data_op)

    _schema_op = schema_op(
        dsl.InputArgumentPath(_stat_op.outputs['stats'])
    ).after(_stat_op)

    _val_op = val_op(
        dsl.InputArgumentPath(_stat_op.outputs['stats']),
        dsl.InputArgumentPath(_schema_op.outputs['schema'])
    ).after(_stat_op, _schema_op)

    _preprocess_op = preprocess_op(
        dsl.InputArgumentPath(_load_data_op.outputs['data'])
    ).after(_load_data_op, _val_op)

    _rf_op = rf_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['clean_data'])
    ).after(_preprocess_op)

    _keras_op = keras_op(
```

```

        dsl.InputArgumentPath(_preprocess_op.outputs['clean_data']))
    ).after(_preprocess_op)
    _lr_op = lr_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['clean_data']))
    ).after(_preprocess_op)
    _cb_op = cb_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['clean_data']))
    ).after(_preprocess_op)
    _knn_op = knn_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['clean_data']))
    ).after(_preprocess_op)

    _sv_op = sv_op(
        dsl.InputArgumentPath(_preprocess_op.outputs['clean_data']))
    ).after(_preprocess_op)

    _eval_op = eval_op(
        dsl.InputArgumentPath(_rf_op.outputs['rf_metrics']),
        dsl.InputArgumentPath(_keras_op.outputs['keras_metrics']),
        dsl.InputArgumentPath(_lr_op.outputs['lr_metrics']),
        dsl.InputArgumentPath(_cb_op.outputs['cb_metrics']),
        dsl.InputArgumentPath(_knn_op.outputs['knn_metrics']),
        dsl.InputArgumentPath(_sv_op.outputs['sv_metrics']))
    ).after(_rf_op, _keras_op, _lr_op, _cb_op, _knn_op, _sv_op)

    _push_op = push_op(bucket_name, credentials,
        dsl.InputArgumentPath(_eval_op.outputs['best_model']))
    ).after(_eval_op)

```

Our pipeline goes through loading the data, carrying out descriptive statistics, data validation before processing data for our six models before evaluation of the metrics,

and exporting the best model to the cloud storage. Each step started as a self-contained function with docker images created from them.

This pipeline function can then be compiled into a yaml file, zip and tar.gz formats are also acceptable and then uploaded to the Kubeflow platform.

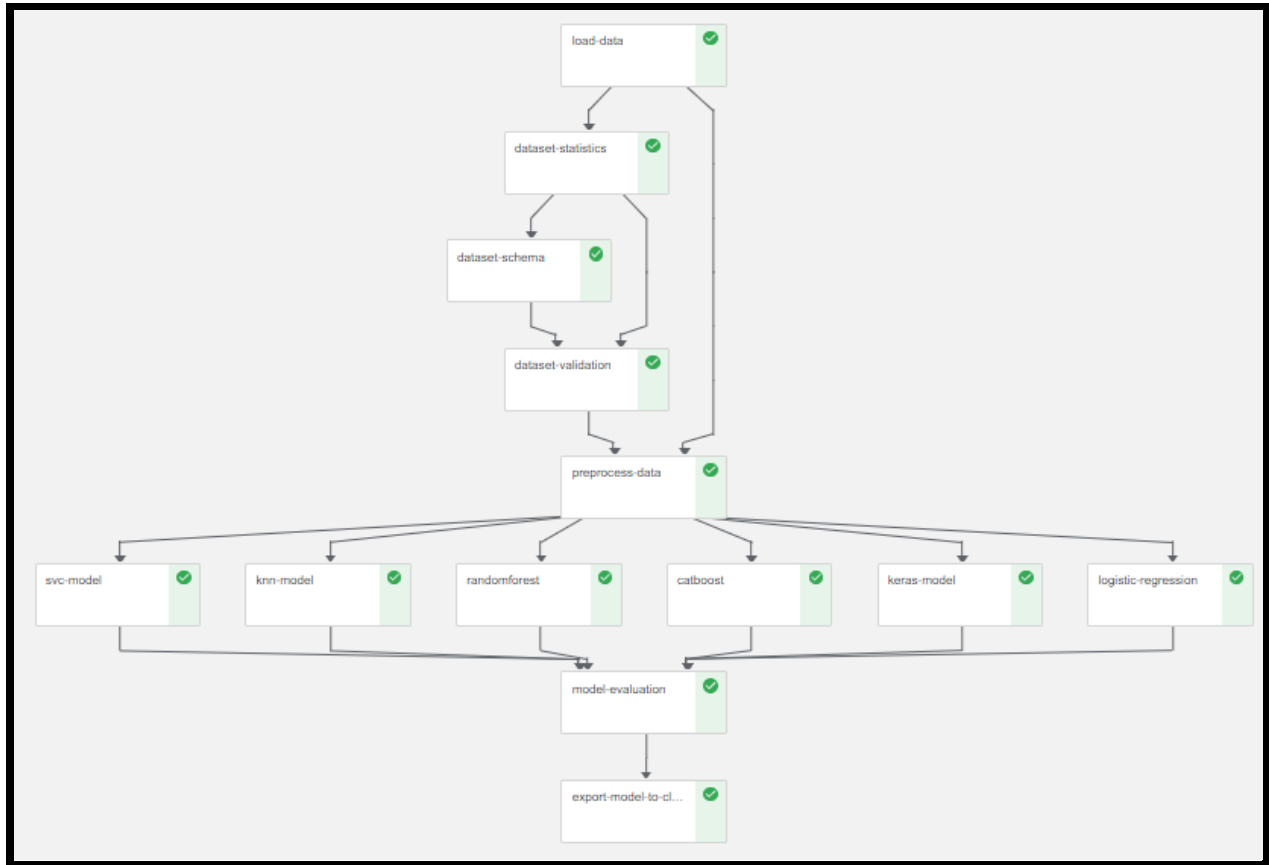
Figure 15: Compiling and Uploading Pipeline

```
# Compile pipeline to generate compressed YAML definition of the
pipeline.
experiment_name = 'heart_pipeline'

kfp.compiler.Compiler().compile(heart_pipeline,
'{}.yaml'.format(experiment_name))

# Client requires an endpoint and credentials to your Kubeflow
client = kfp.Client()
client.create_run_from_pipeline_func(heart_pipeline, arguments={})
```

Figure 16: Kubeflow Pipeline



Conclusion

AI's goal is to make computers and other devices more effective in solving difficult healthcare problems, and by doing so, we can interpret data collected from the diagnosis of chronic diseases such as cardiovascular (heart) diseases. In the same vein, we have applied tools and techniques in machine learning to our heart disease use case to help predict the likelihood of a person having a heart attack or not. We went a step further to make our results available for Android devices for portability and scalability. To this end, early diagnosis of the likelihood of a person having a heart attack with our approach will be very helpful in minimizing complications of the disease.

References

https://www.who.int/health-topics/cardiovascular-diseases/#tab=tab_1
<https://www.cdc.gov/heartdisease/facts.htm>

<https://www.kaggle.com/rashikrahmanpritom/heart-attack-analysis-prediction-dataset>
http://rstudio-pubs-static.s3.amazonaws.com/24341_184a58191486470cab97acdbbfe78ed5.html
<https://docs.seldon.io/projects/alibi/en/latest/methods/ALE.html>
<https://developer.android.com/ml?authuser=1>
<https://www.tensorflow.org/lite/guide?authuser=1>
<https://enterpriseproject.com/article/2019/2/kubernetes-operators-plain-english>
<https://docs.d2iq.com/dkp/kaptain/1.0.1-0.5.0/tutorials/metadata/>
https://www.who.int/health-topics/cardiovascular-diseases/#tab=tab_1
<https://www.heart.org/en/health-topics/heart-attack/understand-your-risks-to-prevent-a-heart-attack>