Report on Hyperparameter Tuning and Model Deployment for Housing Data using Kubernetes Cluster

Project Overview: I tried using the Housing dataset to train a machine-learning model in this project. The approach leverages Kubernetes to run multiple hyperparameter tuning jobs in parallel, identifying the best-performing model, and then deploying that model using a CI/CD pipeline.

The overall goals were:

- Efficiently tune hyperparameters to maximize model performance.
- Create a CI/CD pipeline that automates the entire workflow.
- Deploy the best-performing model into production using Kubernetes.

Initial Approach

The initial strategy was to use Kubernetes for parallel training jobs, Docker for containerizing the application, and GitHub Actions to set up a CI/CD pipeline. Here's a breakdown of key decisions:

1. **Multipass for Local Kubernetes Testing**:
    - Rather than starting with a cloud-based Kubernetes environment, Multipass was a lightweight, local development option. This made testing faster and avoided cloud costs while ensuring Kubernetes configurations were functional before scaling.
2. **Kubernetes for Job Management**:
    - Kubernetes allows for easy deployment and management of containers and supports parallel job execution, making it ideal for hyperparameter tunin.
3. **Docker for Containerization**:
    - Docker allows the packaging of the Python model training code with its dependencies, ensuring consistency across environments.
4. **CI/CD Pipeline with GitHub Actions**:
    - GitHub Actions provided a seamless way to trigger Kubernetes job deployments

Challenges Faced and Solutions

Challenge: Dynamic Job Configuration for Hyperparameter Tuning

Writing individual YAML files for each training job was time-intensive and error-prone. I addressed this by creating a launch_jobs.sh script that dynamically generates and deploys job configurations based on defined hyperparameter ranges. This approach allowed us to scale hyperparameter combinations with minimal manual intervention.

Challenge: Storage for Best Model Across Pods

Since Kubernetes pods are stateless, I implemented a Persistent Volume Claim (PVC) to store model files, allowing different pods to access and save model versions reliably. This ensured that once the best model was selected, it could be stored persistently and accessed by other pods for deployment.

Challenge: Managing Kubernetes Configurations Locally and on GitHub Actions

By using a shared KUBE_CONFIG secret, we were able to configure Kubernetes contexts dynamically in GitHub Actions. This allowed us to develop and test the pipeline locally using Multipass and then deploy it in the CI/CD pipeline with minimal configuration changes. I chose this option because of the time constraint I had, if it was a production I would be more comfortable using Vault Service for storing these crucial credentials.

Challenge: Consistently Identifying the Best Model

To compare models, I created a select_best_model.py script that reads accuracy metrics from each model's output file, selects the model with the highest accuracy, and stores it in a persistent volume. This selection script runs as a separate job in the CI/CD pipeline.

Technical Architecture

Docker Containerization:
All Python scripts, including train.py, select_best_model.py, and serve_model.py, were packaged into Docker images. This ensured that the entire ML environment, including dependencies and model code, remained consistent across different stages of the pipeline.

Kubernetes Orchestration:
Parallel Job Execution: Kubernetes was used to create multiple job instances, each with unique

hyperparameter configurations. Using Kubernetes' Job resource, I ensured that each combination of hyperparameters (e.g., n_estimators and max_depth for Random Forest) ran in a separate, isolated pod.

Persistent Volume Claim (PVC): To maintain access to models across jobs, I utilized a PVC (my-pvc) to store intermediate and final models. This storage solution enabled the sharing of files between Kubernetes pods and facilitated model selection.

Dynamic Job Configuration: launch_jobs.sh generated YAML files dynamically based on the hyperparameter grid, allowing us to launch multiple Kubernetes jobs efficiently.

CI/CD Pipeline Steps

To maintain a continuous and automated flow from training to deployment, I set up a CI/CD pipeline in GitHub Actions with the following stages:

Training Phase:
Hyperparameter Job Generation: Using launch_jobs.sh, we defined grids for n_estimators and max_depth, dynamically creating and applying Kubernetes job files for each configuration.

Kubernetes Job Application: The script then iterated over hyperparameter combinations, applied each job using kubectl apply -f, and launched multiple pods concurrently on Kubernetes.

Model Checkpointing: Each pod saved its trained model and corresponding accuracy file to the shared PVC.

Model Selection:
After all training jobs were complete, we triggered a model selection job using select_best_model.py. This script compared accuracy scores across all model checkpoints to identify the model with the highest performance. The selected model was then tagged as the best and stored for deployment.

Accuracy File Reading: To make the process efficient, the model file structure included an accuracy file that accompanied each model .pkl file, making it easy to read and compare performance metrics.

Deployment Phase:
Kubernetes Deployment: I created a model-deployment.yaml to manage deployment, referencing the best-performing model from PVC storage. serve_model.py was configured to load this model on deployment.

Rollout Check: Using Kubectl rollout status, the CI/CD pipeline confirmed the success of the deployment.

Experimental Setups:

Setup for Kubernetes:

I set up three virtual machine instances using Multipass, which I've been working with for a while due to its simplicity, lightweight design, and close resemblance to production-level remote servers. I created one instance as the master node, named k8s-master, and two worker nodes, named k8s-worker-1 and k8s-worker-2. This setup will allow me to replicate a production-level environment.

Please refer to Multipass documents if you want to know more about this tool

Create Master and worker instances

multipass launch --name k8s-master --cpus 4 --mem 4G --disk 20G
multipass launch --name k8s-worker-1 --cpus 4 --mem 4G --disk 20G
multipass launch --name k8s-worker-2 --cpus 4 --mem 4G --disk 20G

Step 1: Initialize the Master Node

Let's initialize the master node, and let's add and install all the dependencies we need for this setup in the master node. I created a script that includes all the necessary scripts to set a server as the master node
You can find the script on /scripts/ directory.

- **Initialize Kubernetes**:
  On the master node, run the following command to initialize Kubernetes:
  sudo kubeadm init --pod-network-cidr=192.168.0.0/16

- **Set Up kubeconfig**:
  After the master node initializes, set up the kubeconfig for the kubectl command.

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

- **Install a Pod Network Add-On**
  Choose a Network Plugin: After the master node is initialized, you need to install a network add-on. A commonly used one is Calico. You can install it with the following command:
  kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml

- **Get the Master IP:**

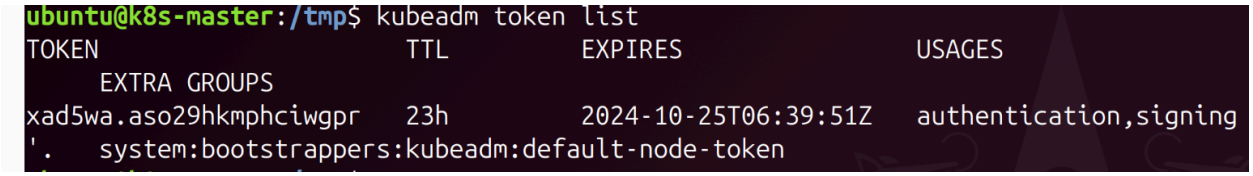
  Identify the Master Node's IP Address:

  On your master node, you can find the IP address by running:
  ip a

  My IP Address was 10.144.158.2

- **Get the Port**
  The default port for the Kubernetes API server is 6443.

- **Get the Join Token**
  List the Available Tokens:
  kubeadm token list



- **Get the Discovery Token CA Cert Hash**

  Get the CA Cert Hash:
  On the Master Node:

  Run the following command to get the correct discovery-token-ca-cert-hash:

  ```
  openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | \
  openssl rsa -pubin -outform der 2>/dev/null | \
  openssl dgst -sha256 -hex | sed 's/^.* //'
  ```

  This will give you the correct hash (32 bytes long), something like:
  sha256:891a78572a1017fce92fe6a46bc914688d94c659fb4e28374c48e4d8f0

Set the worker nodes

Use the same script as for the master node from /script/

Join Worker Nodes

Get Join Command: After the master node initialization, you will see a command to join the worker nodes. It will look something like this:

kubeadm join <master-ip>:<port> --token <token> --discovery-token-ca-cert-hash <hash>

Run Join Command on Worker Nodes: Log into each worker node and run the join command you received from the master node.

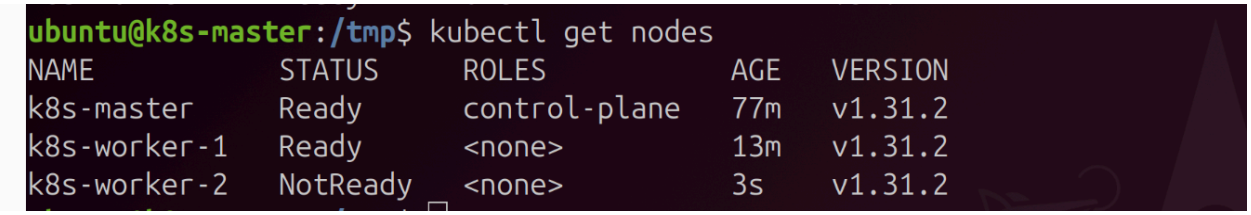
After adding both workers

Verify Cluster Status

Once all nodes have joined, verify the cluster setup from the master node:

kubectl get nodes

Verify using get nodes command



Set Up Pod Network (Master Node Only)

We need to set up the pod network so that pods can communicate across nodes.

- **Install a pod network (e.g., Calico):**
  kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
  This configures the pod network with the CIDR you specified (192.168.0.0/16) during cluster initialization.
- **Verify the pod network is up:**
  kubectl get pods --all-namespaces
  You should see the calico-node and related pods running in the output.

```
ubuntu@k8s-master:/tmp$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                       READY   STATUS             RESTARTS         AGE
kube-system   calico-kube-controllers-6879d4fcdc-47trq   1/1     Running            0                86m
kube-system   calico-node-98jkw                          0/1     Running            10 (5m28s ago)   86m
kube-system   calico-node-wvxhs                          0/1     Running            2 (35s ago)      10m
kube-system   calico-node-zfz5r                          0/1     CrashLoopBackOff   7 (61s ago)      23m
kube-system   coredns-7c65d6cfc9-lvv6l                   1/1     Running            0                86m
kube-system   coredns-7c65d6cfc9-ww9ng                   1/1     Running            0                86m
kube-system   etcd-k8s-master                            1/1     Running            0                88m
kube-system   kube-apiserver-k8s-master                  1/1     Running            0                88m
kube-system   kube-controller-manager-k8s-master         1/1     Running            3 (3m22s ago)    88m
kube-system   kube-proxy-cf5sg                           0/1     CrashLoopBackOff   7 (3m38s ago)    23m
kube-system   kube-proxy-gb4m9                           0/1     CrashLoopBackOff   17 (2m53s ago)   86m
kube-system   kube-proxy-rjrmt                           1/1     Running            4 (50s ago)      10m
```

Deploy Test Application

Once the worker nodes have successfully joined the cluster, you can deploy a test application on the master node to verify everything works.

1. **Deploy a simple Nginx application:**
   kubectl create deployment nginx --image=nginx
2. **Expose the Nginx deployment to access it externally:**
   kubectl expose deployment nginx --port=80 --type=NodePort
3. **Verify the Nginx service is running:**
   kubectl get services
4. **Access Nginx from any browser:** Open your browser and navigate to http://<worker-node-ip>:<NodePort> (you can find the NodePort from the output of the

   kubectl get services

   command).
5.

```
k8s-worker-2   Ready    <none>          19m   v1.31.2
ubuntu@k8s-master:/tmp$ kubectl get nodes
NAME           STATUS   ROLES           AGE   VERSION
k8s-master     Ready    control-plane   97m   v1.31.2
k8s-worker-1   Ready    <none>          33m   v1.31.2
k8s-worker-2   Ready    <none>          19m   v1.31.2
ubuntu@k8s-master:/tmp$ kubectl scale deployment nginx --replicas=3
deployment.apps/nginx scaled
ubuntu@k8s-master:/tmp$ kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS   AGE     IP               NODE           NOMINATED NO
nginx-676b6c5bbc-b994z   1/1     Running   0          6m51s   192.168.140.1    k8s-worker-2   <none>
nginx-676b6c5bbc-ntzcg   1/1     Running   0          6m51s   192.168.140.2    k8s-worker-2   <none>
nginx-676b6c5bbc-rxsv2   1/1     Running   0          7m18s   192.168.230.1    k8s-worker-1   <none>
ubuntu@k8s-master:/tmp$
```

Access the Service

Once you have the node's IP address, you can access the service by opening a browser or using curl to visit the node's IP address followed by the port number 32719:

http://<node-ip>:32719

```
ubuntu@k8s-master:/tmp$ kubectl get services
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP        100m
nginx        NodePort    10.109.8.109   <none>        80:32719/TCP   10m
ubuntu@k8s-master:/tmp$ kubectl get nodes -o wide
NAME           STATUS   ROLES           AGE    VERSION   INTERNAL-IP
NTIME
k8s-master     Ready    control-plane   102m   v1.31.2   10.144.158.2
/1.7.12
k8s-worker-1   Ready    <none>          37m    v1.31.2   10.144.158.80
/1.7.12
k8s-worker-2   Ready    <none>          24m    v1.31.2   10.144.158.220
/1.7.12
ubuntu@k8s-master:/tmp$
```

Access Kubernetes Dashboard:

1. **Install the Dashboard**:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.6.0/aio/deploy/recommended.yaml
```

Access the Dashboard:

To access the dashboard, you'll need to create a service account and a cluster role binding:

```
kubectl create serviceaccount dashboard-admin -n kubernetes-dashboard
kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin
--serviceaccount=kubernetes-dashboard:dashboard-admin
```

Then get the token to log in:

```
kubectl get secret -n kubernetes-dashboard $(kubectl get serviceaccount dashboard-admin -n
kubernetes-dashboard -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 --decode
```

Finally, access the Dashboard by running:

```
kubectl proxy
```

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/