National College of Ireland

Configuration Manual

MSc Research Project

Data Analytics

Rohan Vijay Dongare

Student ID: x18120199

School of Computing

National College of Ireland

Supervisor: Prof. Noel Cosgrave

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Rohan..Vijay…Dongare……

Student ID: …..X18120199…….……

Programme: ....MSc. Data Analytics……. **Year:** …2019…

Module: ……Research..Project……..

Supervisor: …….Prof. Noel Cosgrave…..

Submission Due Date: …….12th August, 2019…….

Project Title: …Star-Galaxy-Quasar Classification using a Novel Stacked Generalization Technique….

Word Count: …… **Page Count**……6…….…..……..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……

Date: ……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Contents

List of Figures

Configuration Manual

Rohan Vijay Dongare
Student ID: x18120199

1 Machine Specifications

The specifications of google cloud instance used in this research are,

1. GPU Specifications:
   13 GB GDDR5 VRAM
   GPU chip: 1x Nvidia Tesla K80
   Cores: 2496 CUDA cores

2. Operating System: Linux
   OS release: 4.14.79+

3. CPU Specifications:
   Intel Xeon Processor (1 core, 2 threads)
   Clock rate: 2.3 GHz
   Cache Available: 45 MB
   Machine: x86_64
   Processor: x86_64
   byteorder: little

4. Disk Space: 320 GB

5. Python Version: 3.6.8
   python-bits: 64
   (Tensorflow has compatibility issues with Python 3.7.x versions, so Google Colaboratory currently runs on the latest 3.6.x release)

6. Tensorflow version: 1.14.0

All the tasks in this study were used on the GPU runtime type as it was more effective than the CPU. Developers at google have tested the GPU vs CPU implementation in colab and find that GPU can offer up to 23x speed up over CPU implementation[1]. The default runtime environment is set to the local system and needs to be changed to GPU before execution of the code.

Versions of the installed libraries are given in figure 1.

Libraries in addition to the ones specified in figure 1 that require installation are

1. pymrmr – 0.1.8
2. nimfa – 1.3.4
3. factor-analyzer- 0.3.1
4. git python – 2.1.12
5. hyperas – 0.4.1
6. hyperopt – 0.1.2
7. hpsklearn.- 0.1.0

[1] https://colab.research.google.com/notebooks/gpu.ipynb

```
pandas: 0.24.2
pytest: 3.6.4
pip: 19.2.1
setuptools: 41.0.1
Cython: 0.29.13
numpy: 1.16.4
scipy: 1.3.0
pyarrow: 0.14.1
xarray: 0.11.3
IPython: 5.5.0
sphinx: 1.8.5
patsy: 0.5.1
dateutil: 2.5.3
pytz: 2018.9
blosc: None
bottleneck: 1.2.1
tables: 3.4.4
numexpr: 2.6.9
feather: 0.4.0
matplotlib: 3.0.3
openpyxl: 2.5.9
xlrd: 1.1.0
xlwt: 1.3.0
xlsxwriter: None
lxml.etree: 4.2.6
bs4: 4.6.3
html5lib: 1.0.1
sqlalchemy: 1.3.6
pymysql: None
psycopg2: 2.7.6.1 (dt dec pq3 ext lo64)
jinja2: 2.10.1
s3fs: 0.3.1
fastparquet: None
pandas_gbq: 0.4.1
pandas_datareader: 0.7.4
gcsfs: None
```

Figure 1: Python Library versions

2 Collecting Data from the Sloan Digital Sky Survey

The Sloan Digital Sky Survey provides a SQL based backend to access their data for research purposes and the data is hosted on the CasJobs server[2]. After creating an account on the Sky Server[3], you can then log in to your personal account where you can store your personal data in their SQL database.
The context tab in Query provides Databases of all the SDSS releases from DR1 to DR15, select DR15 in this tab, by default it will be set to your own database. A table name is then to be specified in the case of this study the table name specified was 'x18120199_research_project'. It was important for objects to not just have photometric values but also to be spectroscopically confirmed as the study was interested in both photometric and spectroscopic values. CasJobs holds two different tables for photometric and spectroscopic values and using a JOIN command, objects matching from both the tables can be retrieved to your table.

Query used for this research is given below,

[2] http://skyserver.sdss.org/CasJobs/jobdetails.aspx?id=43949412&message=Details%20of%2043949412
[3] https://apps.sciserver.org/login-portal/register

SELECT 2000000
r.objid, r.ra, r.dec, r.u, r.g, r.r, r.i, r.z, r.run, r.rerun, r.camcol, r.field,
d.specobjid, d.class, d.z as redshift, d.plate, d.mjd, d.fiberid into mydb.x18120199_research_project from PhotoObj AS r
JOIN SpecObj AS d ON d.bestobjid = r.objid

After the Query is submitted the page redirects to 'My Details' page, that gives status of the query. When the status changes value from 'Ready' to 'Finished' head on to your personal database. A table download option is available, a simple query: SELECT * FROM x18120199_research_project, will start the download of the table to your local system.

3 Importing Data into Google Collaboratory

Since Google colab was used in this study, data could not directly be used from the local system. There are two approaches of using the data from local system. The google.colab package offers a files library, using which data can be uploaded directly to your local workspace.

```
from google.colab import files
uploadFiles = files.upload()
```

This works well if the data is to be used for a single time however the files from the local workspace get deleted at the end of every session. For that reason, the files were uploaded to github first and then called in the colab workspace. Due to the size restrictions of github, Git Large File Storage(Git-LFS) was used. Installation of Git-LFS using Git bash is explained in the following steps:

1. Open git bash in the directory where the file is present.
2. Initialize the directory using 'git init' command
3. Connect to your local directory using the command:
   git remote add origin "https://github.com/Account name/Repository.git"
4. Before pushing new files, pull all the existing files into your directory:
   git pull origin master
5. Check the status of the files in the directory using 'git status'
6. Install lfs using the command:
   git lfs install
7. Since the file to be uploaded was a csv file a tracker for .csv needs to be set first:
   git lfs track "*.csv"
8. The attributes for the path name then need to be set using:
   git add .gitattributes
9. The files can then be committed and pushed like normal files.

4 Bayesian Optimization for the Base Learners

Bayesian optimization for all the base learners in this research was carried out using the hyperopt library. Certain parameters of the sklearn models however cannot be tuned using the hyperopt library. A hyperopt wrapper for the sklearn package was used for tuning all the base learners except Neural Networks.

The package can be installed and used using the following steps:

1. Install gitpython in your workspace
2. Clone the hyperopt-sklearn repository using:
   Repo.clone_from("https://github.com/hyperopt/hyperopt-sklearn", "hyperopt/hyperopt-sklearn.git")
3. Once the folder is cloned to your local workspace navigate to the setup files:
   cd hyperopt/hyperopt-sklearn.git/
4. Install all the setup files using the command:
   pip install -e .

5 Bayesian Optimization for Neural Networks

Bayesian optimization for the Neural Networks was carried out using a keras compatible wrapper for hyperopt called Hyperas. Hyperas can be installed directly using the 'pip install' command.
Tuning a neural network with hyperas requires three important steps. The first step is to call a data function that should return training and testing samples, second step is calling a model function which contains model parameters that require tuning and the final step is calling the execution function.

1. Data Function: It is mandatory to have this data function to successfully execute Bayesian optimization. A snippet of the data function used for the research is attached below. All the pre-processing that was required before splitting the data was carried out in this function itself. A snippet of the function can be seen in figure 1.

```
def under_data():
    import keras
    from keras import utils as cat_utils
    from sklearn.preprocessing import LabelEncoder, PowerTransformer
    from keras.models import Sequential
    import time
    from keras.layers import Dense
    from keras.layers import Dropout
    from keras.layers import Activation
    #fetching raw data from my own repository
    data_git = 'https://raw.githubusercontent.com/rohandongare-nci/'\
    '18120199-Data/master/Base%20Learner%20Optimization/Base%20Learners%20Optimization.csv'
    sdss_opt_nn = pd.read_csv(data_git)
    unwanted_columns = ['camcol','run','rerun','objid','specobjid']
    sdss_opt_nn.drop(unwanted_columns, axis=1, inplace=True)
    enc = LabelEncoder()
    X_opt_nn = sdss_opt_nn[['ra','dec','u','g','r','i','z','redshift','plate','mjd','fiberid','field']]
    y_opt_nn = enc.fit_transform(sdss_opt_nn['class'])
    #normalizing the data
    X_opt_nn = preprocessing.normalize(X_opt_nn, norm='l2')
    #transforming the data using yeo-johnson transformer
    transform_opt = preprocessing.PowerTransformer(method='yeo-johnson', standardize=True)
    X_opt_nn = transform_opt.fit_transform(X_opt_nn)
    under_sampler_opt = RandomUnderSampler(random_state=18120199,replacement=False)
    X_undersam_nn, y_undersam_nn = under_sampler_opt.fit_resample(X_opt_nn, y_opt_nn)
    #onehot encoding the target variable
    y_undersam_nn = cat_utils.to_categorical(y_undersam_nn, 3)
    xtrain, xval, ytrain, yval = train_test_split(X_undersam_nn,y_undersam_nn,test_size=0.2, random_state=18120199)
    return xtrain, ytrain, xval, yval
```

Figure 2: Data Function Hyperas

2. Defining the Model Function:
   A model function then needs to be called which contains all the hyper-parameters that require tuning. The hyperas architecture is built on top of the keras architecture so all the parameters defined in keras were tuned in this study. The number of neurons, activation function, optimizer and the learning rate were tuned using the 'choice' function provided by hyperas. Continuous values like the dropout rate were tuned using the 'uniform' function. The number of layers can also be tuned using hyperas using a simple conditional statement along with the 'choice' function which contains the number of layers. The parameters of the extra layer were also tuned. The snippet of the model function can be seen in figure 2. This model function is later called in the execution function of hyperas.

```
def model_opt(xtrain, ytrain, xval, yval):
    model_sdss_opt = Sequential()
    #defining search space for number of neurons
    model_sdss_opt.add(Dense({{choice([64,128,256,512,1024])}}, input_shape=(12,)))
    #defining search space for the activation function using the choice function
    model_sdss_opt.add(Activation({{choice(['relu','tanh','sigmoid'])}}))
    #defining search space for dropout regularization
    model_sdss_opt.add(Dropout({{uniform(0, 1)}}))
    #similarly defining search space for layer 2,3,4
    model_sdss_opt.add(Dense({{choice([64,128,256,512,1024])}}))
    model_sdss_opt.add(Activation({{choice(['relu','tanh','sigmoid'])}}))
    model_sdss_opt.add(Dropout({{uniform(0, 1)}}))
    model_sdss_opt.add(Dense({{choice([64,128,256,512,1024])}}))
    model_sdss_opt.add(Activation({{choice(['relu','tanh','sigmoid'])}}))
    model_sdss_opt.add(Dropout({{uniform(0, 1)}}))
    model_sdss_opt.add(Dense({{choice([64,128,256,512,1024])}}))
    model_sdss_opt.add(Activation({{choice(['relu','tanh','sigmoid'])}}))
    model_sdss_opt.add(Dropout({{uniform(0, 1)}}))
    #checking if an additional layer will lead to a better performance
    if {{choice(['four', 'five'])}} == 'five':
        model_sdss_opt.add(Dense({{choice([64,128,256,512,1024])}}))
        model_sdss_opt.add(Activation({{choice(['relu', 'sigmoid','tanh'])}}))
        model_sdss_opt.add(Dropout({{uniform(0, 1)}}))
    model_sdss_opt.add(Dense(3))
    model_sdss_opt.add(Activation('softmax'))
    adam_opt = keras.optimizers.Adam(lr={{choice([10**-4, 10**-3, 10**-2])}})
    model_sdss_opt.compile(loss='categorical_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    model_sdss_opt.fit(xtrain, ytrain,
              batch_size={{choice([150,300])}},
              nb_epoch=1,
              verbose=1,
              validation_data=(xval, yval))
    eval_score, model_acc = model_sdss_opt.evaluate(xval, yval, verbose=1)
    print('test accuracy acheived:', model_acc)
    return {'loss': -model_acc, 'status': STATUS_OK, 'model': model_sdss_opt}
```

Figure 3: Model function for hyperas

3. Execution function: Along with the data function and model function the execution function requires the entire python notebook to be passed as well. The python notebook should be saved to the local workspace and the name should be specified in 'notebook_name' parameter. To use hyperas with google colab firstly the entire file needs to be downloaded to your local machine. Then it needs to be uploaded evry time during the execution of the code using the file upload command from google.colab.
   from google.colab import files
   uploadFiles = files.upload()
   Hyperas currently provides only the minimize function so basically the parameters that find the smallest value of a set evaluation metric will be returned by this function. This study was however interested in getting more objects accurately classified and hence the accuracy metric was chosen. While passing the accuracy metric, negative value of it was passed for the minimize function as it will minimize the metric provided.

```
start_nn=time.time()
best_nn_run, best_nn_model = optim.minimize(model=model_opt,
                                            data=under_data,
                                            algo=tpe.suggest,
                                            max_evals=5,
                                            trials=Trials(),
                                            notebook_name='Bayesian_final')
end_nn=time.time()
```

Figure 4: Execution function for Hyperas

6 Bayesian Optimization for the Meta learner

The meta learner was trained using a novel stacking approach that combines both two sampling techniques (K-fold and holdout sampling. A separate sample of data first needs to be uploaded to github for optimizing the meta learner. The predictions of the base learners were then taken using the stacking function. A snippet of the stacking function is given in figure 4.

```
def get_stack_pred(classifier):
  #Initial train-test split- train sample will be used for k-fold, test sample will be used as the holdout set
  x_train, x_test, y_train, y_test = train_test_split(X_under, y_under, test_size = 0.25, random_state = 18120199)
  #initializing arrays to store predictions
  if_test=np.empty((x_train.shape[0],))
  oof_kfold = np.empty((k, x_test.shape[0]))
  infoldx_train=[]
  infoldy_train=[]
  infold_test=[]
  infoldy_test=[]
  #assigning train-test indices for every fold
  for j, (train_index, test_index) in enumerate(str_kf.split(x_train, y_train)):
    infoldx_train = x_train[train_index]
    infoldy_train = y_train[train_index]
    infoldx_test   = x_train[test_index]
    infoldy_test  = y_train[test_index]
    #print(infoldx_test.shape[0])
    #training the classifier on the infold train/test set
    classifier.fit(infoldx_train,infoldy_train)
    #infold test prediction
    if_test[test_index]=classifier.predict(infoldx_test)
    infold_pred=classifier.predict(infoldx_test)
    #Infold classifier performance
    infold_accuracy = metrics.accuracy_score(infold_pred, infoldy_test)
    print(f"In Fold classifier accuracy: {infold_accuracy}")
    #holdout fold test prediction
    pred2=classifier.predict(x_test)
    #holdout set classifier performance
    out_of_fold_accuracy = metrics.accuracy_score(pred2,y_test)
    print(f"Out Of Fold accuracy: {out_of_fold_accuracy}")
    #adding predictions of the holdout set for every fold row-wise to an array
    oof_kfold[j,:]=classifier.predict(x_test)
  #taking mode of the holdout set predictions of each row
  mode=stats.mode(oof_kfold,axis=0)
  oof_kfold=mode[0]
  '''returning transposed columns so that predictions of each base learner
  will later after appending them will have its own column'''
  return if_test.reshape(-1,1), oof_kfold.reshape(-1,1)
```

Figure 5: Stacking Function.

Predictions of the k-fold cross validation as well as the test set of all the base learners were then appended to two separate arrays. The meta learner was trained using the k-fold predictions and validated using the test predictions. Hyperas was used for optimizing the meta learner as well. Section 5 covers the three essential steps for running Bayesian optimization using hyperas. The issue faced in optimizing the meta learner was that as mentioned in section 5, it needs a data function. All the values outside the function are not recognized by the hyperas execution function. Implementing the entire stacked model in a single function complicated the entire process. So, the appended arrays that contained the predictions of the base learners were converted to csv files first. The csv files get saved to your local workspace; they can be downloaded from there using the files library.

from google.colab import files

files.download('filename.csv')

The files then need to be uploaded to a github repository which can be called directly from the data function. A snippet of the data function for optimizing the meta learner is given in figure 5.

```
def meta_opt_data():
    import keras
    from keras import utils as cat_utils
    from sklearn.preprocessing import LabelEncoder, PowerTransformer
    from keras.models import Sequential
    import time
    from keras.layers import Dense
    from keras.layers import Dropout
    from keras.layers import Activation
    import pandas as pd
    #base learner input
    #base learner output
    train_meta = 'https://raw.githubusercontent.com/rohandongare-nci/18120199-Data/master/New-Base-learner-data/train_meta_base_op.csv'
    y_train='https://raw.githubusercontent.com/rohandongare-nci/18120199-Data/master/New-Base-learner-data/y_train.csv'
    test_meta='https://raw.githubusercontent.com/rohandongare-nci/18120199-Data/master/New-Base-learner-data/test_meta_base_op.csv'
    y_test='https://raw.githubusercontent.com/rohandongare-nci/18120199-Data/master/New-Base-learner-data/y_test.csv'
    xtrain = pd.read_csv(train_meta)
    ytrain=pd.read_csv(y_train)
    xval=pd.read_csv(test_meta)
    yval=pd.read_csv(y_test)
    ytrain = cat_utils.to_categorical(ytrain, 3)
    yval = cat_utils.to_categorical(yval, 3)
    return xtrain, ytrain, xval, yval
```

Figure 6: Data function for optimizing the meta learner

The model function and the execution function work exactly like the process described in section 5.