# Project_Proposal

## Introduction

The dataset used for this project is the 1.88 Million US Wildfires dataset.

The dataset was obtained via Kaggle using the link https://www.kaggle.com/datasets/rtatman/188-million-us-wildfires?datasetId=2478&sortBy=voteCount.

This data set provides information on wildfires in America between 1992 and 2015 including when the wildfire was discovered, when the wildfire was contained, who reported the wildfire, where the wildfire occurred and the cause of the wildfire. The objective for this project is a multi-classification problem to predict the cause of a wildfire given the above information.

## Exploratory Data Analysis

Our original data set contains 38 features, with many missing values, categorical features with too many classes, and highly correlated features. These features will be problematic when we fit our models later, hence will be removed. First the features with ~50% of its observations missing and any features that are unique identifiers such as FOD_ID and OBJECTID were removed. Moreover, categorical features with more than 20 levels were removed, although some of these will be revisited when building our final models. We will also look into supplemental sources of data such as temperature and weather data. The features labelled FIRE_SIZE_CLASS, STATE and COUNTY, OWNER_DESCR were removed since these values can be determined from the FIRE_SIZE, LATITUDE, LONGITUDE and OWNER_CODE respectively. The features FIRE_YEAR and CONT_DOY are highly correlated with features DISCOVERY_DATE and DISCOVERY_DOY respectively, hence were removed. The feature CONT_DUR which is the number of days it took to contain the wildfire, was created from CONT_DATE. Lastly the Shape feature was removed since it wasn't useful in predicting the class of the wildfire. There were two more columns, STAT_CAUSE_CODE and STAT_CAUSE_DESCR, for our response variable. However, STAT_CAUSE_CODE was a unique numerical identifier for STAT_CAUSE_DESCR (cause of fire) and hence was dropped. The column STAT_CAUSE_DESCR will be used as our response variable.

There are 13 classes in our original response variable and 1,880,465 observations in our original dataset. However, there is still duplicated and missing data in our dataset both of which were removed to avoid redundancy. The large amount of data ensured there is sufficient data remaining. The classes miscellaneous and missing/undefined, do not provide us with the cause of the wildfire and so were removed. The final dataset contains 9 features and 751,420 observations. Furthermore, the response variable contains 11 classes.

Since the classes were heavily imbalanced, this would have resulted in the classifier rarely ever predicting the least common classes. To ensure that all classes have enough data to fit our models we decided each class in our training set should have the same number of observations. However, keeping in mind the limited computational resources (used later for hyperparameter tuning and cross-validation), we settled on a value of 10,000 observations per class. This was accomplished using both random oversampling and random undersampling (through the imblearn package). The random oversampling procedure starts by selecting all classes with less than 10,000 observations and through simple random sampling with replacement, observations from the class are sampled and added to the dataset until the minority classes reach 10,000 observations. Then by the random under sampling process for each class with more than 10,000 data points,

observations are randomly selected with equal probability to be removed until the number of observations for all classes is 10,000. We are considering changing the amount from 10,000 for the final models we build.

## Classification Methods

To analyse the data, five different classifications methods will be used. These methods are multinomial logistic regression, k-nearest neighbours, support vector machines, gradient-boosted decision trees, and neural networks. The models we will be using will be trained on a common training set and assessed against each other using a common test set. This common training and test set will be formed by splitting the dataset into portions of 90% and 10% respectively (using sklearn's preprocessing module). However, we are considering increasing the testing set size in the final project to make better use of our dataset. Furthermore, the class balancing mentioned in the previous paragraph will be performed on the training set here. This will result in our training set having 110,000 observations and 9 features.The libraries used, the tuning and model parameters calibration process, the model selection process, and final model assessment process for each of the four models will vary depending on the classification method. However, to compare between the final models from each of the four methods, each model will be primarily assessed using prediction accuracy on the common holdout test set. A confusion matrix, ROC-AUC and F1-scores will also be used to assess how well our models perform on classifying specific classes.

For K-nearest neighbours (hereafter referred to as KNN), we will analyse weighted KNN models. Since KNN does not work well with categorical features, only continuous features will be used. Weighted KNN, is similar to regular KNN except that each of the K neighbours are given a weight ("Weighted -KNN") determined via a kernel function such that neighbours further away from the estimated point have lower weight ("Weighted -KNN"). The code will be done in R using KernelKnn and caret packages. The tuning parameters are the number of neighbours (K) and the kernel function, such as rectangular (regular KNN), Gaussian, triangular and Epanechinikov kernels. To calibrate the tuning parameters, 5 fold cross validation will be used with the loss function being the prediction accuracy. The final KNN model selected will be the one with the lowest cross validation error.

For multinomial logistic regression, the coding will be done in R using the nnet, glmnet and caret packages. Multinomial logistic regression models with interaction terms will be considered. Models fitted using LASSO, Ridge and elastic net will also be considered. There are two tuning parameters in this model, the shrinkage parameter lambda and alpha. For the shrinkage parameter, the default parameter from the glmnet package will be tested. The alpha values that will be tested will range from zero to one increasing by 0.1 each time. For tuning and model parameter calibration, 5-fold cross validation will be used. The loss function that will be used for the 5-fold cross validation will be the prediction accuracy. The final logistic model selected is the one with the smallest cross validation error. For support vector machines the models will be built in Python using the sklearn library for all necessary functions related to creating the models. Models using one vs all and one vs one will be built. In total 2 models will be built, one using one vs all and the other using one vs one. Cross-validation using 5-folds will be performed for model selection, and the model selected using cross-validation will be trained on the training data. Various values for C will be used for tuning. The kernel functions used for tuning will be linear, poly, rbf, and sigmoid. The preset methods "scale" and "auto" will be used for tuning gamma. For the SVM model built for the proposal only the rbf kernel was used, the C values used for tuning were [0.1, 1, 100], "scale" and "auto" were used for tuning gamma, and "None" and "Balanced" were used for tuning the class weight. However, tuning parameters for class weight are unnecessary since we balanced the classes for training using resampling methods and thus will not be tuned for the final model.

For XG boosting, a combination of the sklearn and xgboost library will be used. XGBoost stands for Extreme Gradient Boost, which is a gradient boosted decision tree algorithm. Similar to random forests, XGBoost also contains multiple decision trees. However, the main difference is in the way these trees are generated. While random forests use a concept called bagging, Gradient boosting is, "an extension of boosting where the process of additively generating weak models is formalised as a gradient descent algorithm over an objective function. Gradient boosting sets targeted outcomes for the next model in an effort to minimize errors.

Targeted outcomes for each case are based on the gradient of the error (hence the name gradient boosting) with respect to the prediction." ("What Is XGBoost?")

The calibration of tuning parameters was mainly done using sklearn's GridSearchCV. Due to computational constraints, the main parameters that were tuned were max_depth (depth of tree) and n_estimators (number of trees in the forest). More parameters such as learning_rate, column subsampling parameters, and regularization terms will be tuned for the final project with the help of Bayesian Optimization (python library HyperOpt). The final model parameters were selected based on the best performing model (with respective hyperparameters). Then the final model was trained on the entire training dataset with the best performing hyper-parameters (from above step) and the final test accuracy was calculated based on the test set (split originally using train_test_split).

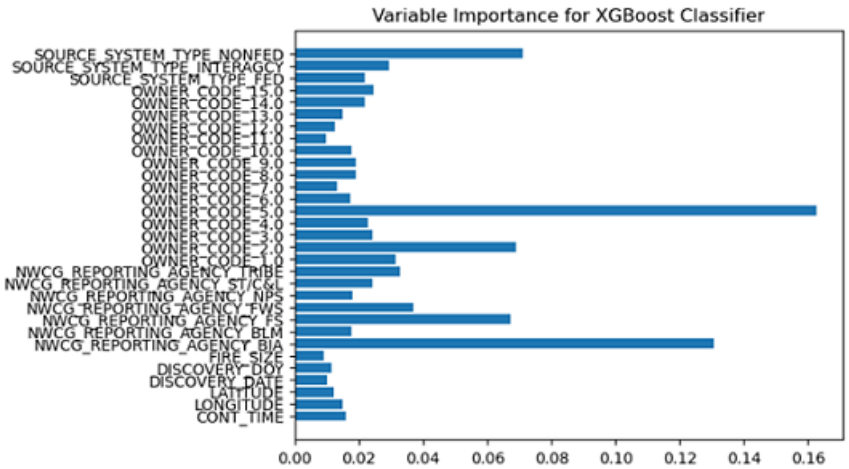
For neural networks, the sklearn.preprocessing module was used for data cleaning. For building and fitting the model, tensorflow.keras will be used. To perform model selection, keras.tuner will be used to tune the number of hidden layers, the number of units in each layer, and the activation function(tanh, sigmoid, etc.). "The Keras Tuner library helps you pick the optimal set of hyperparameters for your TensorFlow program. The process of selecting the right set of hyperparameters for your machine learning (ML) application is called hyperparameter tuning or hypertuning" (Waderkar). Keras Tuner provides multiple options for tuning algorithms, wherein we utilise the RandomSearch tuner. Since the hyperparameter space is extremely large, RandomSearch picks a specific number of combinations based on the 'max_trials' parameter to reduce the number of parameter combinations trained. The optimization algorithm to find the best tuning parameter is the adam algorithm. The loss function that will be used with the adam algorithm is the negative-log likelihood. The best model from all trials will be selected to calibrate the final model.

## Preliminary results

These are the preliminary results for the best performing model of each classifier type

Table 1: Preliminary Results

| | Best_Training_Score_Percent | Test_Score_Percent | Calibration_Time_Mins |
|---|---|---|---|
| Support Vector Machines | 37.8 | 42.2 | 905 |
| XGBoost | 58.3 | 52.9 | 93 |
| Neural Network | 61.5 | 45.0 | 313 |



# Appendix

## Support Vector Machine

```
#Load Libraries
import pandas as pd
import numpy as np
import sqlite3 as sqlite3
import numpy as np
import pickle
from sklearn.model_selection import train_test_split
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
from sklearn.metrics import accuracy_score
from sklearn import svm
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.multiclass import OneVsRestClassifier
```

```
# Load the data into a DataFrame
con = sqlite3.connect("data/wildfire.sqlite")
fires = pd.read_sql_query(
    "select NWCG_REPORTING_AGENCY,CONT_DATE - DISCOVERY_DATE as CONT_TIME, CONT_DOY, \
    LONGITUDE,LATITUDE, SOURCE_SYSTEM_TYPE,DISCOVERY_DATE, FIRE_YEAR,\
    DISCOVERY_DOY,STAT_CAUSE_DESCR,FIRE_SIZE from fires", con)
con.close()
```

```
fires = fires.drop_duplicates()
fires.shape
```

```
fires = fires.dropna()
fires.shape
```

```
fires1 = fires[(fires["STAT_CAUSE_DESCR"] != "Missing/Undefined") &
               (fires["STAT_CAUSE_DESCR"] != "Miscellaneous")]
fires1.shape
```

```
# fires1=fires1.drop(['CONT_DOY', 'FIRE_YEAR', 'combined_date_dis',
#                                              'combined_date_con'],axis=1)
fires1=fires1.drop(['CONT_DOY', 'FIRE_YEAR'],axis=1)
fires1.info()
```

```
xFires=fires1.loc[:,fires1.columns != 'STAT_CAUSE_DESCR']
yFires=fires1['STAT_CAUSE_DESCR']
xFires=pd.get_dummies(xFires, columns=['NWCG_REPORTING_AGENCY', 'SOURCE_SYSTEM_TYPE'])
#Training and test set split
xTrain,xTest,yTrain,yTest=train_test_split(xFires,yFires,\
                          test_size=0.1,random_state =441)
```

```
xTrain.shape
# yTrain.shape
```

```
counts = yTrain.value_counts()
counts
```

```
from imblearn.over_sampling import RandomOverSampler
def count_under_10000(colname):
    if counts[colname] < 10000:
        return 10000
    return counts[colname]
#Perform undersampling
OverSampleRatio = {
    'Lightning' : count_under_10000('Lightning'),
    'Debris Burning' : count_under_10000('Debris Burning'),
    'Campfire' : count_under_10000('Campfire'),
    'Equipment Use' : count_under_10000('Equipment Use'),
    'Arson' : count_under_10000('Arson'),
    'Children' : count_under_10000('Children'),
    'Railroad' : count_under_10000('Railroad'),
    'Smoking' : count_under_10000('Smoking'),
    'Powerline' : count_under_10000('Powerline'),
    'Fireworks' : count_under_10000('Fireworks'),
    'Structure' : count_under_10000('Structure')
    }
#Goal balance all classes
newSampStrat=RandomOverSampler(sampling_strategy=OverSampleRatio,random_state=441)
#perform the balancing newX and newY are balanced X and y
xTrain,yTrain=newSampStrat.fit_resample(xTrain,yTrain)
```

```
counts = yTrain.value_counts()
```

```
from imblearn.under_sampling import RandomUnderSampler
#Goal balance all classes
newSampStrat=RandomUnderSampler(sampling_strategy='not minority',random_state=441)
#perform the balancing newX and newY are balanced X and y
xTrain,yTrain=newSampStrat.fit_resample(xTrain,yTrain)
```

```
# RBF kernel with covariate scaling
model_rbf = Pipeline(
    steps=[("scaler", StandardScaler()),
           ("model", svm.SVC(kernel="rbf"))]
)
model_to_set = OneVsRestClassifier(model_rbf)
# tuning parameter grid
# model__xyz specifies that parameter xyz is a parameter to model
param_grid = {
    "model__C": [.01, 1, 100],
    "model__class_weight": [None, "balanced"],
    "model__gamma": ["scale", "auto"]
}
# crossvalidation folds
```

```
cv = KFold(
    n_splits=5,  # number of folds
    shuffle=True # protects against data being ordered, e.g., all successes first
)
cv_rbf_onevall = GridSearchCV(
    estimator = model_rbf,
    param_grid = param_grid,
    cv = cv
)
```

```
# %%time
cv_rbf_onevall.fit(X=xTrain, y=yTrain)
```

```
filename='SVM_model_proposal'
cv_rbf_onevall = pickle.load(open(filename, 'rb'))
cv_rbf_onevall.cv_results_
```

```
final_model = cv_rbf_onevall.best_estimator_
```

```
yPred = final_model.predict(xTest)
```

```
accuracy = accuracy_score(yTest, yPred)
print("Test Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
print("The best training accuracy score is ", cv_rbf_onevall.best_score_ * 100, "%")
```

## XGBoost

```
# # Need to limit number of cores used
# import os
# os.environ['MKL_NUM_THREADS'] = '1'
```

```
#Load Libraries
import pandas as pd
import sqlite3 as sqlite3
import numpy as np
from matplotlib import pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.metrics import accuracy_score
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
from xgboost import XGBClassifier
```

## Data Preparation

```
# Load the data into a DataFrame
con = sqlite3.connect("data/wildfire.sqlite")
fires = pd.read_sql_query(
    "select NWCG_REPORTING_AGENCY,CONT_DATE - DISCOVERY_DATE as CONT_TIME, \
    LONGITUDE,LATITUDE,OWNER_CODE,SOURCE_SYSTEM_TYPE,DISCOVERY_DATE,\
    DISCOVERY_DOY,STAT_CAUSE_DESCR,FIRE_SIZE from fires", con)
con.close()
```

```
fires.info()
```

```
fires = fires.drop_duplicates()
fires.shape
```

```
fires = fires.dropna()
fires.shape
```

```
fires1 = fires[(fires["STAT_CAUSE_DESCR"] != "Missing/Undefined") & \
               (fires["STAT_CAUSE_DESCR"] != "Miscellaneous")]
fires1.shape
```

```
fires1["STAT_CAUSE_DESCR"].value_counts().plot.bar()
```

```
xFires=fires1.loc[:,fires1.columns != 'STAT_CAUSE_DESCR']
yFires=fires1['STAT_CAUSE_DESCR']
#Training and test set split
xTrain,xTest,yTrain,yTest=train_test_split(xFires,yFires,\
                          test_size=0.1,random_state =441)
xTrain.shape
# yTrain.shape
```

```
counts = yTrain.value_counts()
counts
```

```
from imblearn.over_sampling import RandomOverSampler
def count_under_10000(colname):
    if counts[colname] < 10000:
        return 10000
    return counts[colname]
#Perform undersampling
OverSampleRatio = {
    'Lightning' : count_under_10000('Lightning'),
    'Debris Burning' : count_under_10000('Debris Burning'),
    'Campfire' : count_under_10000('Campfire'),
    'Equipment Use' : count_under_10000('Equipment Use'),
    'Arson' : count_under_10000('Arson'),
    'Children' : count_under_10000('Children'),
    'Railroad' : count_under_10000('Railroad'),
    'Smoking' : count_under_10000('Smoking'),
    'Powerline' : count_under_10000('Powerline'),
    'Fireworks' : count_under_10000('Fireworks'),
    'Structure' : count_under_10000('Structure')
```

```
    }
#Goal balance all classes
newSampStrat=RandomOverSampler(sampling_strategy=OverSampleRatio,random_state=441)
#perform the balancing newX and newY are balanced X and y
xTrain,yTrain=newSampStrat.fit_resample(xTrain,yTrain)
#print result showing the nunmber of observation in each class
yTrain.value_counts().plot.bar()
```

```
counts = yTrain.value_counts()
counts
```

```
from imblearn.under_sampling import RandomUnderSampler
#Goal balance all classes
newSampStrat=RandomUnderSampler(sampling_strategy='not minority',random_state=441)
#perform the balancing newX and newY are balanced X and y
xTrain,yTrain=newSampStrat.fit_resample(xTrain,yTrain)
yTrain.value_counts().plot.bar()
```

```
yTrain.value_counts()
```

## Feature Engineering

```
xTrain.info()
```

```
xTrain.DISCOVERY_DATE
```

```
#list for cols to scale
cols_to_scale = ['CONT_TIME','LONGITUDE', 'LATITUDE', 'DISCOVERY_DATE', \
                 'DISCOVERY_DOY', "FIRE_SIZE"]
#create and fit scaler
scaler = StandardScaler()
scaler.fit(xTrain[cols_to_scale])
#scale selected data
xTrain[cols_to_scale] = scaler.transform(xTrain[cols_to_scale])
```

```
# #create and fit scaler
# scaler = StandardScaler()
# scaler.fit(xTest[cols_to_scale])
#scale selected data
xTest[cols_to_scale] = scaler.transform(xTest[cols_to_scale])
```

```
# OHE for categorical variables
xTrain_model = pd.get_dummies(xTrain, columns = ["NWCG_REPORTING_AGENCY",\
                                                 "OWNER_CODE", "SOURCE_SYSTEM_TYPE"])
xTrain_model.info()
```

```
xTest_model = pd.get_dummies(xTest, columns = ["NWCG_REPORTING_AGENCY",\
                                               "OWNER_CODE", "SOURCE_SYSTEM_TYPE"])
# xTest_model.info()
```

```
# keeping only same columns as xTrain
cols_to_keep = [col for col in xTest_model.columns if col in xTrain_model.columns]
xTest_model = xTest_model[cols_to_keep]
xTest_model.info()
```

```
# Encode classes
lc = LabelEncoder()
lc_yTrain = lc.fit_transform(yTrain)
lc_yTrain
lc_yTest = lc.transform(yTest)
lc_yTest
```

```
lc.get_params()
```

## Model Training

```
params = {
        #'gamma': [0.3, 1],
        'max_depth': [40, 100],
        'n_estimators' : [200, 500],
#         'learning_rate' : [0.01, 0.1, 0.9],
        'random_state' : [441],
        'reg_alpha' : [0.5]
        }
# params = {
#          'min_child_weight': [1, 5, 10],
#          'gamma': [0.5, 1, 1.5, 2, 5],
#          'subsample': [0.6, 0.8, 1.0],
#          'colsample_bytree': [0.6, 0.8, 1.0],
#          'max_depth': [3, 4, 5],
#          'n_estimators' : [100, 300]
#          }
```

```
estimator = XGBClassifier(objective = "multi:softprob")
grid_search = GridSearchCV(
    estimator=estimator,
    param_grid=params,
    scoring = 'accuracy',
    #n_jobs = ,
    cv = 5,
    verbose=4,
    refit = True
)
grid_search.fit(xTrain_model, lc_yTrain)
# model.fit(xTrain_model, lc_yTrain)
# yPred = model.predict(xTest)
# yPred = [round(value) for value in yPred]
# accuracy = accuracy_score(lc_yTest, yPred)
# print("Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
grid_search.cv_results_
```

```
print("The best training accuracy score is ", grid_search.best_score_ * 100, "%")
```

```
print("The best parameter combination is:\n ", grid_search.best_params_)
```

```
final_model = grid_search.best_estimator_
```

```
# Predict accuracy on the final test set
yPred = final_model.predict(xTest_model)
yPred = [round(value) for value in yPred]
accuracy = accuracy_score(lc_yTest, yPred)
print("Test Accuracy: %.2f%%" % (accuracy * 100.0))
```

```
plt.barh(xTrain_model.columns, final_model.feature_importances_)
plt.title('Variable Importance for XGBoost Classifier')
```

## Neural Network

## Data Preprocessing

```
#Load Libraries
import pandas as pd
import sqlite3 as sqlite3
import numpy as np
from sklearn.model_selection import train_test_split
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
```

```
# Load the data into a DataFrame
con = sqlite3.connect("wildfire.sqlite")
fires = pd.read_sql_query(
    "select NWCG_REPORTING_AGENCY,CONT_DATE - DISCOVERY_DATE as CONT_TIME, \
    LONGITUDE,LATITUDE,OWNER_CODE,SOURCE_SYSTEM_TYPE,DISCOVERY_DATE,\
    DISCOVERY_DOY,STAT_CAUSE_DESCR,FIRE_SIZE from fires", con)
con.close()
```

```
fires.info()
```

```
fires = fires.drop_duplicates()
fires.shape
```

```
fires = fires.dropna()
fires.shape
```

```
fires1 = fires[(fires["STAT_CAUSE_DESCR"] != "Missing/Undefined")& \
               (fires["STAT_CAUSE_DESCR"] != "Miscellaneous")]
fires1.shape
```

```
fires1["STAT_CAUSE_DESCR"].value_counts().plot.bar()
```

```
xFires=fires1.loc[:,fires1.columns != 'STAT_CAUSE_DESCR']
yFires=fires1['STAT_CAUSE_DESCR']
```

## One-hot-encode and Scale the Input

```
from sklearn.compose import ColumnTransformer, make_column_selector
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```
# creat the column transformer(Look into Sparsity Later)
ct = ColumnTransformer([
    ('onehot',
        OneHotEncoder(drop="first"),
        make_column_selector(dtype_include=object)),
    ('scale',StandardScaler(),
        make_column_selector(dtype_include=np.number))],
    verbose_feature_names_out=False)
```

```
xFires = ct.fit_transform(xFires)
ct.get_feature_names_out()
```

```
#Training and test set split
xTrain,xTest,yTrain,yTest=train_test_split(xFires,yFires,\
                          test_size=0.1,random_state =441)
xTrain.shape
# yTrain.shape
```

## Oversampling and Undersampling to cope with imbalance classes

```
counts = yTrain.value_counts()
counts
```

```
from imblearn.over_sampling import RandomOverSampler
def count_under_10000(colname):
    if counts[colname] < 10000:
        return 10000
    return counts[colname]
#Perform undersampling
OverSampleRatio = {
    'Lightning' : count_under_10000('Lightning'),
    'Debris Burning' : count_under_10000('Debris Burning'),
```

```
    'Campfire' : count_under_10000('Campfire'),
    'Equipment Use' : count_under_10000('Equipment Use'),
    'Arson' : count_under_10000('Arson'),
    'Children' : count_under_10000('Children'),
    'Railroad' : count_under_10000('Railroad'),
    'Smoking' : count_under_10000('Smoking'),
    'Powerline' : count_under_10000('Powerline'),
    'Fireworks' : count_under_10000('Fireworks'),
    'Structure' : count_under_10000('Structure')
    }
#Goal balance all classes
newSampStrat=RandomOverSampler(sampling_strategy=OverSampleRatio,random_state=441)
#perform the balancing newX and newY are balanced X and y
xTrain,yTrain=newSampStrat.fit_resample(xTrain,yTrain)
#print result showing the nunmber of observation in each class
yTrain.value_counts().plot.bar()
```

```
counts = yTrain.value_counts()
counts
```

```
from imblearn.under_sampling import RandomUnderSampler
#Goal balance all classes
newSampStrat=RandomUnderSampler(sampling_strategy='not minority',random_state=441)
#perform the balancing newX and newY are balanced X and y
xTrain,yTrain=newSampStrat.fit_resample(xTrain,yTrain)
yTrain.value_counts().plot.bar()
```

```
yTrain.value_counts()
```

```
xTrain.shape
```

```
# check that mean and variance of LONGITUDE is 0/1
print("mean(LONGITUDE) = {}, std(LONGITUDE) = {}".format(
    np.mean(xTrain[:,12]),
    np.std(xTrain[:,12])))
```

## Label the response as numerical values using LabelEncoder

```
from sklearn.preprocessing import LabelEncoder
# Encode the response "STAT_CAUSE_DESCR"
# (probability better just use the encoding from the orginal dataset)
# But want to learn to use sklearn.preprocessing.LabelEncoder
le=LabelEncoder()
yTrain=le.fit_transform(yTrain)
list(le.classes_)
```

```
yTest=le.fit_transform(yTest)
list(le.classes_)
```

```
# Separating the xTrain_trans further into train and validation data
xTrain,xVal,yTrain,yVal=train_test_split(xTrain, yTrain,\
                                         test_size=0.1,random_state=441)
```

## One-hot-encoding response for NN Multiclass Classification

```
from tensorflow import keras
num_classes = 11
yTrain = keras.utils.to_categorical(yTrain, num_classes)
yVal = keras.utils.to_categorical(yVal, num_classes)
#yTest = keras.utils.to_categorical(yTest, num_classes)
```

## Building a Neural Network Model

```
# #!pip install keras-tuner -q (already downloaded)
```

```
import keras_tuner
from tensorflow.keras import layers
```

```
Hyperparameters that could be tuned:
    number of layers,
    number of unit for each layer,
    activation function of each layer,
    learning_rate,
    loss? maybe
    dropout layer? (unknown)
    ...
```

```
# Build model with hyperparameters
def build_model():
    model = keras.Sequential()

    #input layer
    model.add(layers.Dense(units=128, activation='relu', input_shape = (xTrain.shape[1],)))

    #hidden layer(s)
    model.add(layers.Dense(units=128, activation="relu"))

    model.add(layers.Dense(units=128, activation="relu"))

    model.add(layers.Dense(units=128, activation="relu"))

    model.add(layers.Dense(units=128, activation="relu"))

    model.add(layers.Dense(units=128, activation="relu"))

    #Output Layer
```

```
    model.add(layers.Dense(11,activation='softmax'))

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.0001),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model

model=build_model()
model.summary()
```

```
# %%time
EPOCHS = 500
history = model.fit(
        xTrain,
        yTrain,
        epochs=EPOCHS,
        verbose=1,
        shuffle=True,
        validation_data = (xVal, yVal),
    )
```

```
model.predict(xTest)
```

```
predict_classes = np.argmax(model.predict(xTest), axis=1)
```

```
print(f'Test score is: {sum(predict_classes==yTest)/len(yTest)}')
```

# References


Chollet, François, and Others. Keras. 2015, https://keras.io.

Short, Karen C. 'Spatial Wildfire Occurrence Data for the United States, 1992-2015 [FPA_FOD_20170508] (4th Edition)'. Forest Service Research Data Archive, Forest Service Research Data Archive, 2017, https://doi.org/10.2737/RDS-2013-0009.4.

Note - Link is broken, so please refer to kaggle link for dataset https://www.kaggle.com/datasets/rtatman/188-million-us-wildfires

Wadekar, Shakti. "Hyperparameter Tuning in Keras: Tensorflow 2: With Keras Tuner: RandomSearch, Hyperband. . . " Medium, The Startup, 15 Jan. 2021, [https://medium.com/swlh/hyperparameter-tuning-in-keras-tensorflow-2-with-keras-tuner-randomsearch-hyperband-3e212647778f#:~:text=RandomSearch%20concept%3A,the

"Weighted K-NN." GeeksforGeeks, 7 Apr. 2020, www.geeksforgeeks.org/weighted-k-nn

"What Is XGBoost?" NVIDIA Data Science Glossary, www.nvidia.com/en-us/glossary/data-science/xgboost