PROBLEM STATEMENT

Here we propose to implement a classifier which detect the emotion from the text. The aim of this project is to detect and recognize types of feelings such as anger, disgust, fear, happiness, sadness, and surprise. This can find its application in many of the areas where government can detect an overall emotion index if its citizen. This can greatly help in predictions of many diseases and disorders related to mental health and stress. We aim to develop a classifier which can detect and analyze the text and predict the onset of any emotion non- conducive to a health of a person. Another commercial application of this classifier would be in consumer market to analyze the inclination of customers towards a specific product or trend. It can help the businesses to take their important decisions based on the data that they receive using this classifier. Content from social media posts like Twitter, Facebook can be also be used to predict the general sentiments in the public for a particular post.

RANDOM FOREST

Random Forest is combination of many small decision trees. Each decision tree will predict the sample to be belonging to some class. The class which gets most votes from all the decision trees is predicted to be the class for that sample. Not all features are used in a decision tree to predict about the class of the sample. Random features are selected amongst the total feature space. This decision of choosing the feature affects the decision of the tree and must be done cautiously. This reduction in the feature space of the random forest tree helps in speeding up of the learning of the classifier. It can be considered as a bootstrapping algorithm and runs efficiently on large datasets due to its speed. The forest generated are reusable on other kind.

The practical implementation of the algorithm can be jotted down as follows.

- Introduce two sources of randomness: *Bagging* and *Random* input vectors.
- *Bagging*- creating ensembles by bootstrap aggregation- repeated random sub-sampling of the training data.
- *Bootstrap sample* - will on average contain 63.2% of the data while the rest are replicates.
- Using bootstrap sample, a decision tree is grown to its greatest depth minimizing the loss function.
- At each node, best split of decision tree is chosen from random sample of input variables instead of all variables.
- For each tree, using the leftover (36.8%) data, calculate the misclassification rate = out of bag (OOB) error rate.
- Aggregate error from all trees to determine overall OOB error rate for the classification

CONFUSION MATRIX

A table describing the performance of a classifier. This table is build using results from the values of which the true classes are already known. This table will basically contain the numbers when the classifier correctly identified the class and the number when it identified a sample in the wrong class. Confusion Matrix helps in identifying the types of errors made by the classifier which helps in gaining insight into the performance of Classifier and improving it further.

Following terminologies are used in Confusion Matrix is used for comparing the performance of the classifier.

- True positives (TP)
- True negatives (TN)
- False positives (FP)
- False negatives (FN)

PSEUDO CODE

Each tree is constructed using the following algorithm:

- Let the number of training cases be N, and the number of variables in the classifier be M.
- We are told the number m of input variables to be used to determine the decision at a node of the tree; m should be much less than M.
- Choose a training set for this tree by choosing n times with replacement from all N available training cases (i.e. take a bootstrap sample). Use the rest of the cases to estimate the error of the tree, by predicting their classes.
- For each node of the tree, randomly choose m variables on which to base the decision at that node. Calculate the best split based on these m variables in the training set.
- Each tree is fully grown and not pruned (as may be done in constructing a normal tree classifier).
- For prediction, a new sample is pushed down the tree. It is assigned the label of the training sample in the terminal node it ends up in. This procedure is iterated over all trees in the ensemble, and the average vote of all trees is reported as random forest prediction.

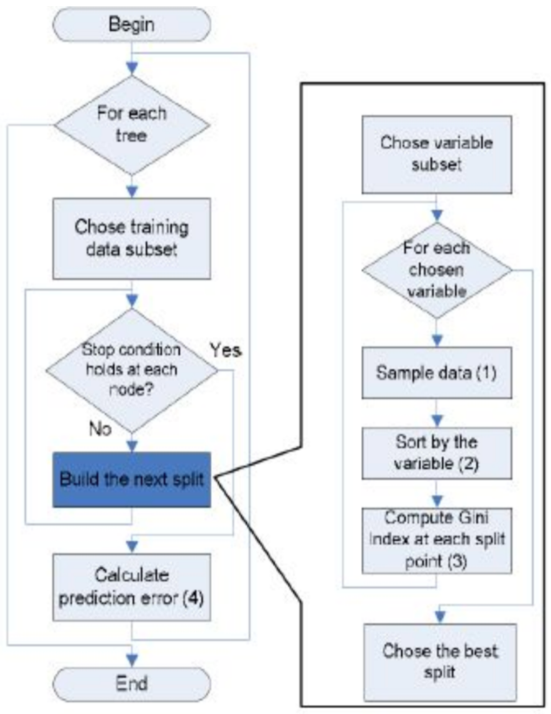

Fig 1. Flowchart for visualization of the algorithm

FUNCTION DEFINITIONS

CLASS	**PARAMETERS**	**DESCRIPTION**
DecisionTree		
	min_samples_split	The minimum number of samples needed to make a split when building a tree.
	min_impurity	The minimum impurity required to split the tree further.
	max_depth	The maximum depth of a tree.
	loss	Loss function that is used for Gradient Boosting models to calculate impurity.
ASSOCIATED FUNCTIONS		
buildtree()		Recursive method which builds out the decision tree and splits X and respective y on the feature of X which (based on impurity) best separates the data
predict_value()		Do a recursive search down the tree and make a prediction of the data sample by the value of the leaf that we end up at
predict()		Classify samples one by one and return the set of labels
print_tree()		Recursively print the decision tree

CLASS		
RandomForest		Random Forest classifier. Uses a collection of classification trees that trains on random subsets of the data using a random subset of the features.
	n_estimators	The number of classification trees that are used.

	max_features	The maximum number of features that the classification trees are allowed to use.
	min_samples_split	The minimum number of samples needed to make a split when building a tree.
	main_gain	The minimum impurity required to split the tree further.
	max_depth	The maximum depth of a tree.
ASSOCIATED FUNCTIONS		
fit(), predict()		Fit the algorithm for the dataset
confusion_matrix()		Creation of Confusion matrix for evaluation of the algorithm
divide_on_feature()		Divide dataset based on the threshold for sample value on feature index
calculate_entropy()		Calculate the entropy of the label array
get_random_subsets()		Return random subsets of the data for with replacements

CLASS		
ClassificationTree	Decision Tree	
ASSOCIATED FUNCTIONS		
_calculate_information_gain		Calculate Information Gain
_majority_vote		Count number of occurences of samples with label

DATA DESCRIPTION

- **Human Activity Recognition database**

The dataset is built from the recordings of 30 subjects performing activities of daily living (ADL) while carrying a waist-mounted smartphone with embedded inertial sensors. Each person performed six activities (WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING) wearing a smartphone (Samsung Galaxy S II) on the waist. Using its embedded accelerometer and gyroscope, we captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz. The experiments have been video-recorded to label the data manually. The obtained dataset has been randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data. The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of features was obtained by calculating variables from the time and frequency domain. The dataset contains split sets for training and testing. We leverage this and the technique of encoding the categorical dataset and using them to train the system.

For each record, we have -

1. Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
2. Triaxial Angular velocity from the gyroscope.
3. A 561-feature vector with time and frequency domain variables.
4. Activity labels.
5. An identifier of the subject who carried out the experiment

Data set Characteristics	Multivariate, Time Series
Associated Tasks	Classification, Clustering
Number of instances	10299
Number of attributes	561

- **Bank Marketing Dataset**

The data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution. The classification goal is to predict if the client will subscribe a term deposit (variable y). The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed.

For each record, we have:

- Age (numeric)
- job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management','retired','self-employed','services','student','technician','unemployed','unknown')
- marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
- education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate','professional.course','university.degree','unknown')
- default: has credit in default? (categorical: 'no','yes','unknown')
- housing: has housing loan? (categorical: 'no','yes','unknown')
- loan: has personal loan? (categorical: 'no','yes','unknown')
  # related with the last contact of the current campaign:
- contact: contact communication type (categorical: 'cellular','telephone')
- month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
- day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')
- duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.
- campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
- pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
- previous: number of contacts performed before this campaign and for this client (numeric)
- poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent','success')
- emp.var.rate: employment variation rate - quarterly indicator (numeric)
- cons.price.idx: consumer price index - monthly indicator (numeric)
- cons.conf.idx: consumer confidence index - monthly indicator (numeric)
- euribor3m: euribor 3 month rate - daily indicator (numeric)
- nr.employed: number of employees - quarterly indicator (numeric)

Desired target

- Classifying the dataset into two classes; predict if the client between 21 - y - has subscribed a term deposit.

Data set Characteristics	Multivariate
Associated Tasks	Classification
Number of instances	4522
Number of attributes	17

EVALUATION AND RESULTS

1. Evaluation of Human Activity Recognition Dataset

	WALKING	WALKING UPSTAIRS	WALKING DOWNSTAIRS	SITTING	STANDING	LAYING
WALKING	454	5	37	0	0	0
WALKING UPSTAIRS	54	405	12	0	0	0
WALKING DOWNSTAIRS	51	54	315	0	0	0
SITTING	0	0	0	383	87	21
STANDING	0	0	0	65	462	4
LAYING	0	0	0	34	6	497

Table 1. Confusion matrix for dataset Human activity analysis

As we can see from the above confusion matrix

- WALKING was predicted as WALKING for 454 sets, as WALKING UPSTAIRS for 54 sets and as downstairs for 51 sets. As these have similar sensor data we can see our system fails to detect 104 sets.
- WALKING UPSTAIRS was predicted as WALKING UPSTAIRS for 405 sets, as WALKING UPSTAIRS for 5 sets and WALKING DOWNSTAIRS for54 sets as. As these have similar sensor data we can see our system fails to detect 59 sets.
- WALKING DOWNSTAIRS was predicted as WALKING DOWNSTAIRS 405 sets, as WALKING DOWNSTAIRS for 37 sets and as WALKING UPSTAIRS for 12 sets. As these have similar sensor data we can see our system fails to detect 49 sets.
- SITTING was predicted as SITTING for 383 sets, as STANDING for 65 sets and as LAYING for 37 sets. As these have similar sensor data we can see our system fails to detect 102 sets.
- STANDING was predicted as STANDING for 462 sets, as SITTING for 87 sets and as LAYING for 6 sets. As these have similar sensor data we can see our system fails to detect 93 sets.
- LAYING was predicted as LAYING for 497 sets, as SITTING for 21 sets and as STANDING for 4 sets. As these have less similar sensor data we can see our system fails to detect 25 sets.

The total accuracy we got for the Human activity recognition is Accuracy = 85.40 %As the dataset was very huge with 600 columns with each column having 7500 data entries for training set and 3000 entries for test set we ran our classifier for 20 estimators (number of classification trees).

The total time taken in a Computer 16 GB RAM and a Intel core i7 was 45mins. We can achieve even high accuracy if we can run the algorithm for higher estimators (number of classification trees).

2. Evaluation of Bank Market Dataset

Estimations=10	Predicted NO	Predicted Yes
Actual NO	TN=988	FP=5
Actual Yes	FN=120	TP=22

Table 2. Confusion matrix for 10 Estimators

Accuracy = 88.98

Estimations =50	Predicted NO	Predicted Yes
Actual NO	TN=989	FP=4
Actual Yes	FN=115	TP=27

Table 2: Confusion matrix for 50 Estimators

Accuracy: 89.42

The total time taken in a Computer 16 GB RAM and a Intel core i7 was 2mins for 10 estimations and 5 mins for 50 estimations. We can achieve even high accuracy if we can run the algorithm for higher estimators (number of classification trees).

<u>REFERENCES</u>

[1] Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research 3.Feb (2003): 1137-1155. [4] Schwenk, Holger. "Continuous space language models." Computer Speech & Language 21.3 (2007): 492-518.

[2] Mikolov, Tomáš, et al. "Empirical evaluation and combination of advanced language modeling techniques." Twelfth Annual Conference of the International Speech Communication Association. 2011.

[3] Pennington, Jeffrey, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

[4] Xu, Baoxun, et al. "An Improved Random Forest Classifier for Text Categorization." JCP 7.12 (2012): 2913-2920.

[5] https://www.stat.berkeley.edu/~breiman/randomforest2001.pdfs

[6] http://www.ijcaonline.org/archives/volume178/number3/vora-2017-ijca-915773.pdf

APPENDIX

IMPLEMENTATION

Human Activity Recognition Dataset.

```
from __future__ import division, print_function
import numpy as np
import math
import progressbar

import matplotlib.pyplot as plt
import pandas as pd


def divide_on_feature(X, feature_i, threshold):
    split_func = None
    if isinstance(threshold, int) or isinstance(threshold, float):
        split_func = lambda sample: sample[feature_i] >= threshold
    else:
        split_func = lambda sample: sample[feature_i] == threshold

    X_1 = np.array([sample for sample in X if split_func(sample)])
    X_2 = np.array([sample for sample in X if not split_func(sample)])

    return np.array([X_1, X_2])

def calculate_entropy(y):

    log2 = lambda x: math.log(x) / math.log(2)
    unique_labels = np.unique(y)
    entropy = 0
    for label in unique_labels:
        count = len(y[y == label])
        p = count / len(y)
        entropy += -p * log2(p)
    return entropy

def get_random_subsets(X, y, n_subsets, replacements=True):

    n_samples = np.shape(X)[0]
    # Concatenate x and y and do a random shuffle
    X_y = np.concatenate((X, y.reshape((1, len(y))).T), axis=1)
    np.random.shuffle(X_y)
    subsets = []

    # Uses 50% of training samples without replacements
    subsample_size = int(n_samples // 2)
    if replacements:
        subsample_size = n_samples      # 100% with replacements

    for _ in range(n_subsets):
        idx = np.random.choice(
            range(n_samples),
            size=np.shape(range(subsample_size)),
            replace=replacements)
        X = X_y[idx][:, :-1]
        y = X_y[idx][:, -1]
        subsets.append([X, y])
    return subsets

bar_widgets = [
    'Training: ', progressbar.Percentage(), ' ', progressbar.Bar(marker="-", left="[", right="]"),
    ' ', progressbar.ETA()
]

class DecisionNode():
    #Class that represents a decision node or leaf in the decision tree

    def __init__(self, feature_i=None, threshold=None,
                 value=None, true_branch=None, false_branch=None):
        self.feature_i = feature_i          # Index for the feature that is tested
        self.threshold = threshold          # Threshold value for feature
        self.value = value                  # Value if the node is a leaf in the tree
        self.true_branch = true_branch      # 'Left' subtree
        self.false_branch = false_branch    # 'Right' subtree



class DecisionTree(object):
    #Super class of RegressionTree and ClassificationTree.


    def __init__(self, min_samples_split=2, min_impurity=1e-7,
                 max_depth=float("inf"), loss=None):
        self.root = None  # Root node in dec. tree
        # Minimum n of samples to justify split
        self.min_samples_split = min_samples_split
        # The minimum impurity to justify split
        self.min_impurity = min_impurity
        # The maximum depth to grow the tree to
        self.max_depth = max_depth
        # Function to calculate impurity (classif.=>info gain, regr=>variance reduct.)
        self._impurity_calculation = None
        # Function to determine prediction of y at leaf
        self._leaf_value_calculation = None
        # If y is one-hot encoded (multi-dim) or not (one-dim)
        self.one_dim = None
        # If Gradient Boost
        self.loss = loss

    def fit(self, X, y, loss=None):
        #Build decision tree
        self.one_dim = len(np.shape(y)) == 1
        self.root = self._build_tree(X, y)
        self.loss=None

    def _build_tree(self, X, y, current_depth=0):
       # Recursive method which builds out the decision tree and splits X and respective y
       # on the feature of X which (based on impurity) best separates the data

        largest_impurity = 0
        best_criteria = None    # Feature index and threshold
        best_sets = None        # Subsets of the data

        # Check if expansion of y is needed
        if len(np.shape(y)) == 1:
            y = np.expand_dims(y, axis=1)

        # Add y as last column of X
        Xy = np.concatenate((X, y), axis=1)

        n_samples, n_features = np.shape(X)

        if n_samples >= self.min_samples_split and current_depth <= self.max_depth:
            # Calculate the impurity for each feature
            for feature_i in range(n_features):
                # All values of feature_i
                feature_values = np.expand_dims(X[:, feature_i], axis=1)
                unique_values = np.unique(feature_values)

                # Iterate through all unique values of feature column i and
                # calculate the impurity
                for threshold in unique_values:
                    # Divide X and y depending on if the feature value of X at index feature_i
                    # meets the threshold
                    Xy1, Xy2 = divide_on_feature(Xy, feature_i, threshold)
                if len(Xy1) > 0 and len(Xy2) > 0:
                    # Select the y-values of the two sets
                    y1 = Xy1[:, n_features:]
                    y2 = Xy2[:, n_features:]

                    # Calculate impurity
                    impurity = self._impurity_calculation(y, y1, y2)

                    # If this threshold resulted in a higher information gain than previously
                    # recorded save the threshold value and the feature
                    # index
                    if impurity > largest_impurity:
                        largest_impurity = impurity
                        best_criteria = {"feature_i": feature_i, "threshold": threshold}
                        best_sets = {
                            "leftX": Xy1[:, :n_features],   # X of left subtree
                            "lefty": Xy1[:, n_features:],   # y of left subtree
                            "rightX": Xy2[:, :n_features],  # X of right subtree
                            "righty": Xy2[:, n_features:]   # y of right subtree
                            }

    if largest_impurity > self.min_impurity:
        # Build subtrees for the right and left branches
        true_branch = self._build_tree(best_sets["leftX"], best_sets["lefty"], current_depth + 1)
        false_branch = self._build_tree(best_sets["rightX"], best_sets["righty"], current_depth + 1)
        return DecisionNode(feature_i=best_criteria["feature_i"], threshold=best_criteria[
                    "threshold"], true_branch=true_branch, false_branch=false_branch)

    # We're at leaf => determine value
    leaf_value = self._leaf_value_calculation(y)

    return DecisionNode(value=leaf_value)


def predict_value(self, x, tree=None):


    if tree is None:
        tree = self.root

    # If we have a value (i.e we're at a leaf) => return value as the prediction
    if tree.value is not None:
        return tree.value

    # Choose the feature that we will test
    feature_value = x[tree.feature_i]

    # Determine if we will follow left or right branch
    branch = tree.false_branch
    if isinstance(feature_value, int) or isinstance(feature_value, float):
            if feature_value >= tree.threshold:
                branch = tree.true_branch
        elif feature_value == tree.threshold:
            branch = tree.true_branch

        # Test subtree
        return self.predict_value(x, branch)

    def predict(self, X):
       # Classify samples one by one and return the set of labels
        y_pred = [self.predict_value(sample) for sample in X]
        return y_pred

    def print_tree(self, tree=None, indent=" "):
       # Recursively print the decision tree
        if not tree:
            tree = self.root

        # If we're at leaf => print the label
        if tree.value is not None:
            print (tree.value)
        # Go deeper down the tree
        else:
            # Print test
            print ("%s:%s? " % (tree.feature_i, tree.threshold))
            # Print the true scenario
            print ("%sT->" % (indent), end="")
            self.print_tree(tree.true_branch, indent + indent)
            # Print the false scenario
            print ("%sF->" % (indent), end="")
            self.print_tree(tree.false_branch, indent + indent)



class ClassificationTree(DecisionTree):
    def _calculate_information_gain(self, y, y1, y2):
        # Calculate information gain
        p = len(y1) / len(y)
        entropy = calculate_entropy(y)
        info_gain = entropy - p * \
            calculate_entropy(y1) - (1 - p) * \
            calculate_entropy(y2)

        return info_gain

    def _majority_vote(self, y):
        most_common = None
        max_count = 0
        for label in np.unique(y):
            # Count number of occurences of samples with label
            count = len(y[y == label])
            if count > max_count:
                most_common = label
                max_count = count
        return most_common

    def fit(self, X, y):
        self._impurity_calculation = self._calculate_information_gain
        self._leaf_value_calculation = self._majority_vote
        super(ClassificationTree, self).fit(X, y)




class RandomForest():
    #Random Forest classifier. Uses a collection of classification trees that
    #trains on random subsets of the data using a random subsets of the features.

    def __init__(self, n_estimators=100, max_features=None, min_samples_split=2,
                 min_gain=0, max_depth=float("inf")):
        self.n_estimators = n_estimators    # Number of trees
        self.max_features = max_features    # Maxmimum number of features per tree
        self.min_samples_split = min_samples_split
        self.min_gain = min_gain            # Minimum information gain req. to continue
        self.max_depth = max_depth          # Maximum depth for tree
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)

        # Initialize decision trees
        self.trees = []
        for _ in range(n_estimators):
            self.trees.append(
                ClassificationTree(
                    min_samples_split=self.min_samples_split,
                    min_impurity=min_gain,
                    max_depth=self.max_depth))

    def fit(self, X, y):
        n_features = np.shape(X)[1]
        # If max_features have not been defined => select it as
        # sqrt(n_features)
        if not self.max_features:
            self.max_features = int(math.sqrt(n_features))

        # Choose one random subset of the data for each tree
        subsets = get_random_subsets(X, y, self.n_estimators)

        for i in self.progressbar(range(self.n_estimators)):
            X_subset, y_subset = subsets[i]
            # Feature bagging (select random subsets of the features)
            idx = np.random.choice(range(n_features), size=self.max_features, replace=True)
            # Save the indices of the features for prediction
            self.trees[i].feature_indices = idx
            # Choose the features corresponding to the indices
            X_subset = X_subset[:, idx]
            # Fit the tree to the data
            self.trees[i].fit(X_subset, y_subset)

    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))
        # Let each tree make a prediction on the data
        for i, tree in enumerate(self.trees):
            # Indices of the features that the tree has trained on
            idx = tree.feature_indices
            # Make a prediction based on those features
            prediction = tree.predict(X[:, idx])
            y_preds[:, i] = prediction

        y_pred = []
        # For each sample
        for sample_predictions in y_preds:
            # Select the most common class prediction
            y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())
        return y_pred


def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0

# Importing the dataset
dataset = pd.read_csv('bank.csv')
#X = dataset.iloc[:, :-1].values
#y = dataset.iloc[:, 4].values


X_train = pd.read_csv('X_train.csv')
X_train = X_train.iloc[:,:].values
y_train = pd.read_csv('y_train.csv')
y_train = y_train.iloc[:,:].values
X_test = pd.read_csv('X_test.csv')
X_test = X_test.iloc[:,:].values
y_test = pd.read_csv('y_test.csv')
y_test = y_test.iloc[:,:].values


# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
# Fitting Random Forest Classification to the Training set
clf = RandomForest(n_estimators=10)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
accurracy = accuracy_metric(y_test, y_pred)
```

Bank Marketing dataset:

```
from __future__ import division, print_function
import numpy as np
import math
import progressbar

import matplotlib.pyplot as plt
import pandas as pd

# Import helper functions
#from mlfromscratch.utils import divide_on_feature, train_test_split, get_random_subsets, normalize
#from mlfromscratch.utils import accuracy_score, calculate_entropy
#from mlfromscratch.unsupervised_learning import PCA
#from decision_tree.py import ClassificationTree
#from mlfromscratch.utils.misc import bar_widgets
#from mlfromscratch.utils import Plot


def divide_on_feature(X, feature_i, threshold):
    split_func = None
    if isinstance(threshold, int) or isinstance(threshold, float):
        split_func = lambda sample: sample[feature_i] >= threshold
    else:
        split_func = lambda sample: sample[feature_i] == threshold

    X_1 = np.array([sample for sample in X if split_func(sample)])
    X_2 = np.array([sample for sample in X if not split_func(sample)])

    return np.array([X_1, X_2])


def calculate_entropy(y):

    log2 = lambda x: math.log(x) / math.log(2)
    unique_labels = np.unique(y)
    entropy = 0
    for label in unique_labels:
        count = len(y[y == label])
        p = count / len(y)
        entropy += -p * log2(p)
    return entropy

def get_random_subsets(X, y, n_subsets, replacements=True):

    n_samples = np.shape(X)[0]
    # Concatenate x and y and do a random shuffle
    X_y = np.concatenate((X, y.reshape((1, len(y))).T), axis=1)
    np.random.shuffle(X_y)
    subsets = []

    # Uses 50% of training samples without replacements
    subsample_size = int(n_samples // 2)
    if replacements:
        subsample_size = n_samples      # 100% with replacements

    for _ in range(n_subsets):
        idx = np.random.choice(
            range(n_samples),
            size=np.shape(range(subsample_size)),
            replace=replacements)
        X = X_y[idx][:, :-1]
        y = X_y[idx][:, -1]
        subsets.append([X, y])
    return subsets

bar_widgets = [
    'Training: ', progressbar.Percentage(), ' ', progressbar.Bar(marker="-", left="[", right="]"),
    ' ', progressbar.ETA()
]

class DecisionNode():
    #Class that represents a decision node or leaf in the decision tree

    def __init__(self, feature_i=None, threshold=None,
                 value=None, true_branch=None, false_branch=None):
        self.feature_i = feature_i          # Index for the feature that is tested
        self.threshold = threshold          # Threshold value for feature
        self.value = value                  # Value if the node is a leaf in the tree
        self.true_branch = true_branch      # 'Left' subtree
        self.false_branch = false_branch    # 'Right' subtree


class DecisionTree(object):
    #Super class of RegressionTree and ClassificationTree.


    def __init__(self, min_samples_split=2, min_impurity=1e-7,
                 max_depth=float("inf"), loss=None):
        self.root = None  # Root node in dec. tree
        # Minimum n of samples to justify split
        self.min_samples_split = min_samples_split
        # The minimum impurity to justify split
        self.min_impurity = min_impurity
        # The maximum depth to grow the tree to
        self.max_depth = max_depth
        # Function to calculate impurity (classif.=>info gain, regr=>variance reduct.)
        self._impurity_calculation = None
        # Function to determine prediction of y at leaf
        self._leaf_value_calculation = None
        # If y is one-hot encoded (multi-dim) or not (one-dim)
        self.one_dim = None
        # If Gradient Boost
        self.loss = loss

    def fit(self, X, y, loss=None):
        #Build decision tree
        self.one_dim = len(np.shape(y)) == 1
        self.root = self._build_tree(X, y)
        self.loss=None

    def _build_tree(self, X, y, current_depth=0):
       # Recursive method which builds out the decision tree and splits X and respective y
       # on the feature of X which (based on impurity) best separates the data

        largest_impurity = 0
        best_criteria = None    # Feature index and threshold
        best_sets = None        # Subsets of the data

        # Check if expansion of y is needed
        if len(np.shape(y)) == 1:
            y = np.expand_dims(y, axis=1)

        # Add y as last column of X
        Xy = np.concatenate((X, y), axis=1)

        n_samples, n_features = np.shape(X)

        if n_samples >= self.min_samples_split and current_depth <= self.max_depth:
            # Calculate the impurity for each feature
            for feature_i in range(n_features):
                # All values of feature_i
                feature_values = np.expand_dims(X[:, feature_i], axis=1)
                unique_values = np.unique(feature_values)

                # Iterate through all unique values of feature column i and
                # calculate the impurity
                for threshold in unique_values:
                    # Divide X and y depending on if the feature value of X at index feature_i
                    # meets the threshold
                    Xy1, Xy2 = divide_on_feature(Xy, feature_i, threshold)
                if len(Xy1) > 0 and len(Xy2) > 0:
                    # Select the y-values of the two sets
                    y1 = Xy1[:, n_features:]
                    y2 = Xy2[:, n_features:]

                    # Calculate impurity
                    impurity = self._impurity_calculation(y, y1, y2)

                    # If this threshold resulted in a higher information gain than previously
                    # recorded save the threshold value and the feature
                    # index
                    if impurity > largest_impurity:
                        largest_impurity = impurity
                        best_criteria = {"feature_i": feature_i, "threshold": threshold}
                        best_sets = {
                            "leftX": Xy1[:, :n_features],   # X of left subtree
                            "lefty": Xy1[:, n_features:],   # y of left subtree
                            "rightX": Xy2[:, :n_features],  # X of right subtree
                            "righty": Xy2[:, n_features:]   # y of right subtree
                            }

    if largest_impurity > self.min_impurity:
        # Build subtrees for the right and left branches
        true_branch = self._build_tree(best_sets["leftX"], best_sets["lefty"], current_depth + 1)
        false_branch = self._build_tree(best_sets["rightX"], best_sets["righty"], current_depth + 1)
        return DecisionNode(feature_i=best_criteria["feature_i"], threshold=best_criteria[
                            "threshold"], true_branch=true_branch, false_branch=false_branch)

    # We're at leaf => determine value
    leaf_value = self._leaf_value_calculation(y)

    return DecisionNode(value=leaf_value)


def predict_value(self, x, tree=None):


    if tree is None:
        tree = self.root

    # If we have a value (i.e we're at a leaf) => return value as the prediction
    if tree.value is not None:
        return tree.value

    # Choose the feature that we will test
    feature_value = x[tree.feature_i]

    # Determine if we will follow left or right branch
    branch = tree.false_branch
    if isinstance(feature_value, int) or isinstance(feature_value, float):
            if feature_value >= tree.threshold:
                branch = tree.true_branch
        elif feature_value == tree.threshold:
            branch = tree.true_branch

        # Test subtree
        return self.predict_value(x, branch)

    def predict(self, X):
       # Classify samples one by one and return the set of labels
        y_pred = [self.predict_value(sample) for sample in X]
        return y_pred

    def print_tree(self, tree=None, indent=" "):
       # Recursively print the decision tree
        if not tree:
            tree = self.root

        # If we're at leaf => print the label
        if tree.value is not None:
            print (tree.value)
        # Go deeper down the tree
        else:
            # Print test
            print ("%s:%s? " % (tree.feature_i, tree.threshold))
            # Print the true scenario
            print ("%sT->" % (indent), end="")
            self.print_tree(tree.true_branch, indent + indent)
            # Print the false scenario
            print ("%sF->" % (indent), end="")
            self.print_tree(tree.false_branch, indent + indent)



class ClassificationTree(DecisionTree):
    def _calculate_information_gain(self, y, y1, y2):
        # Calculate information gain
        p = len(y1) / len(y)
        entropy = calculate_entropy(y)
        info_gain = entropy - p * \
            calculate_entropy(y1) - (1 - p) * \
            calculate_entropy(y2)

        return info_gain

    def _majority_vote(self, y):
        most_common = None
        max_count = 0
        for label in np.unique(y):
            # Count number of occurences of samples with label
            count = len(y[y == label])
            if count > max_count:
                most_common = label
                max_count = count
        return most_common

    def fit(self, X, y):
        self._impurity_calculation = self._calculate_information_gain
        self._leaf_value_calculation = self._majority_vote
        super(ClassificationTree, self).fit(X, y)




class RandomForest():
    #Random Forest classifier. Uses a collection of classification trees that
    #trains on random subsets of the data using a random subsets of the features.

    def __init__(self, n_estimators=100, max_features=None, min_samples_split=2,
                 min_gain=0, max_depth=float("inf")):
        self.n_estimators = n_estimators    # Number of trees
        self.max_features = max_features    # Maxmimum number of features per tree
        self.min_samples_split = min_samples_split
        self.min_gain = min_gain            # Minimum information gain req. to continue
        self.max_depth = max_depth          # Maximum depth for tree
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)

        # Initialize decision trees
        self.trees = []
        for _ in range(n_estimators):
            self.trees.append(
                ClassificationTree(
                    min_samples_split=self.min_samples_split,
                    min_impurity=min_gain,
                    max_depth=self.max_depth))

    def fit(self, X, y):
        n_features = np.shape(X)[1]
        # If max_features have not been defined => select it as
        # sqrt(n_features)
        if not self.max_features:
            self.max_features = int(math.sqrt(n_features))

        # Choose one random subset of the data for each tree
        subsets = get_random_subsets(X, y, self.n_estimators)

        for i in self.progressbar(range(self.n_estimators)):
            X_subset, y_subset = subsets[i]
            # Feature bagging (select random subsets of the features)
            idx = np.random.choice(range(n_features), size=self.max_features, replace=True)
            # Save the indices of the features for prediction
            self.trees[i].feature_indices = idx
            # Choose the features corresponding to the indices
            X_subset = X_subset[:, idx]
            # Fit the tree to the data
            self.trees[i].fit(X_subset, y_subset)

    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))
        # Let each tree make a prediction on the data
        for i, tree in enumerate(self.trees):
            # Indices of the features that the tree has trained on
            idx = tree.feature_indices
            # Make a prediction based on those features
            prediction = tree.predict(X[:, idx])
            y_preds[:, i] = prediction

        y_pred = []
        # For each sample
        for sample_predictions in y_preds:
            # Select the most common class prediction
            y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())
        return y_pred


def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0



# Importing the dataset
dataset = pd.read_csv('bank.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, 16].values



# Encoding categorical data
# Encoding the Independent Variable
from sklearn.preprocessing import LabelEncoder
labelencoder_X = LabelEncoder()
X[:, 1] = labelencoder_X.fit_transform(X[:, 1])
X[:, 2] = labelencoder_X.fit_transform(X[:, 2])
X[:, 3] = labelencoder_X.fit_transform(X[:, 3])
X[:, 4] = labelencoder_X.fit_transform(X[:, 4])
X[:, 6] = labelencoder_X.fit_transform(X[:, 6])
X[:, 7] = labelencoder_X.fit_transform(X[:, 7])
X[:, 8] = labelencoder_X.fit_transform(X[:, 8])
X[:, 10] = labelencoder_X.fit_transform(X[:, 10])
```

```
X[:, 15] = labelencoder_X.fit_transform(X[:, 15])

#onehotencoder = OneHotEncoder(categorical_features = [0])
#X = onehotencoder.fit_transform(X).toarray()
# Encoding the Dependent Variable
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)

X=np.array(X).astype(float)
y=np.array(y).astype(float)

# Splitting the dataset into the Training set and Test set
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)


"""
# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
"""
# Fitting Random Forest Classification to the Training set
clf = RandomForest(n_estimators=10)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)


accuracy = accuracy_metric(y_test,y_pred)
```

N=10	Predicted NO	Predicted Yes
Actual NO	TN=190	FP=11
Actual Yes	FN=24	TP=118

N=50	Predicted NO	Predicted Yes
Actual NO	TN=195	FP=6
Actual Yes	FN=6	TP=136

N=100	Predicted NO	Predicted Yes
Actual NO	TN=	FP=
Actual Yes	FN=	TP=

References:

[1] Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research 3.Feb (2003): 1137-1155. [4] Schwenk, Holger. "Continuous space language models." Computer Speech & Language 21.3 (2007): 492-518.

[2] Mikolov, Tomáš, et al. "Empirical evaluation and combination of advanced language modeling techniques." Twelfth Annual Conference of the International Speech Communication Association. 2011.

[3] Pennington, Jeffrey, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

[4] Xu, Baoxun, et al. "An Improved Random Forest Classifier for Text Categorization." JCP 7.12 (2012): 2913-2920.

[5] https://www.stat.berkeley.edu/~breiman/randomforest2001.pdfs

[6] http://www.ijcaonline.org/archives/volume178/number3/vora-2017-ijca-915773.pdf