The Early Detection of Type 2 Diabetes Using Machine Learning Algorithms

Shridha Rajeswar
duPont Manual High School

Introduction

Type 2 Diabetes

Pancreas **doesn't produce** sufficient amounts of **insulin** to regulate glucose / body is **insulin-resistant → excess glucose in blood**

Credit: fundacionarlosslim.org

Context

380 million
people live with Type 2 Diabetes on a **daily basis**

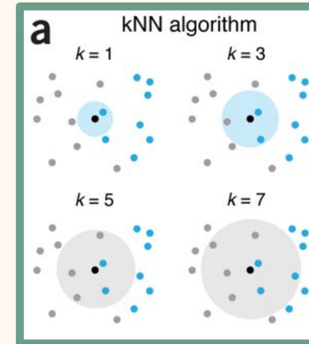
Invasive/Inaccurate
Current diagnosis equipment often produce **false results** (false “positives” and “negatives”)

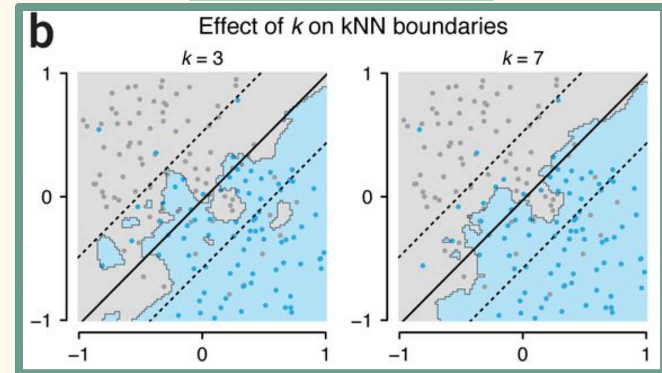
Machine Learning

- ML enables machines to **perform tasks** with **artificial intelligence**
- Multiple types for different purposes
  - **K-Nearest Neighbors (KNN)**
  - **Logistic Regression (LR)**
  - **Decision Tree (DT)**
- Input: diabetic parameter
- Output: diabetic condition

K-Nearest Neighbors

- Used for **classification** and **regression problems**
- Behaves like humans; **affected by** parameters that have **neighboring** or **close values**
  - Data **classified** into different categories **based on proximity** to one another
- Components - **folds** and **nearest neighbors**

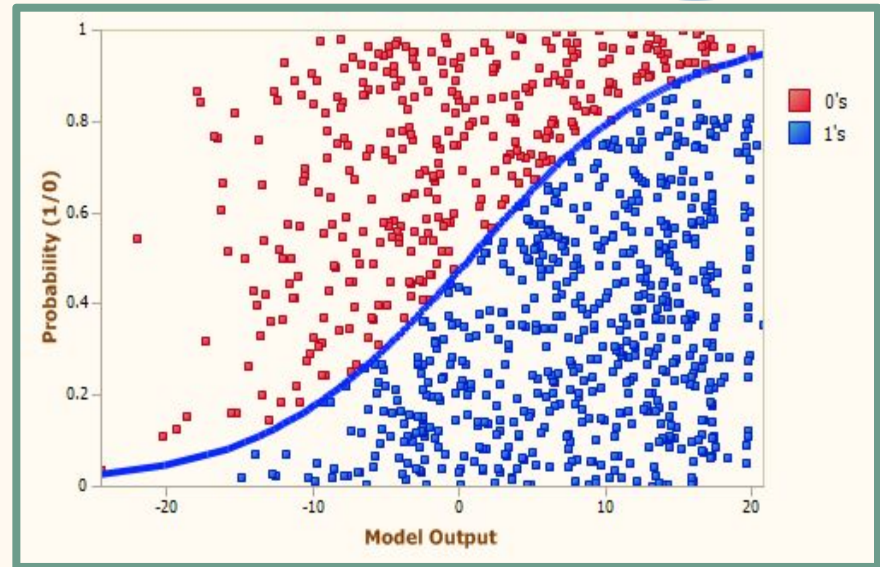




Credit: www.researchgate.net

Logistic Regression

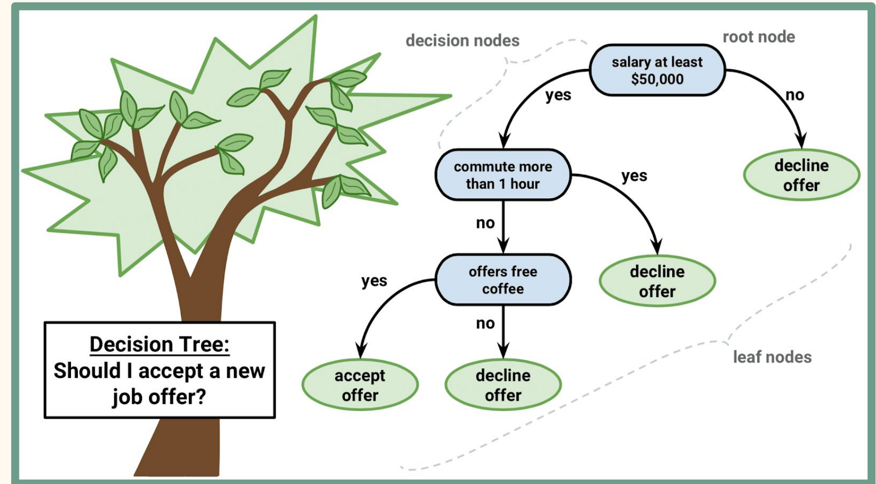
- Used for **regression problems**
- Uses **sigmoid** and **similar functions** to the ones in Artificial Neural Networks; works in **two different phases** (training and testing)
  - **Multiple functions** carried out **to produce output value** from input values
- Components - **learning rate** and **iterations/epochs**



Credit: fusionanalysticsworld.com

Decision Tree

- Used for **classification** and **regression problems**
- Behaves like a tree; continually **splits nodes** at different levels **to categorize input** and **branches off** when close to **predicting outputs**
    - **Gini index** determines **quality of split** to ensure that outputs are accurate
- Components - **maximum depth** and **minimum number of samples for node splitting**



Credit: mediumcom

Engineering Goal

Create **three** different types of ML algorithms that **successfully** predict binary outputs indicating a patient's diabetic condition based on given input values

- Independent variables: **differs** for each ML algorithm type
- Dependent variables: **accuracy rates**
- Constant variables: **folds** (excluding KNN)

KNN

folds, neighbors

LR

learning rate, iterations

DT

max. depth, min. samples for node splitting

Methodology (KNN)

1. Calculate **Euclidean Distance**

Calculate **distance** between two rows in training dataset using ***euclidean_distance()*** **function** ($d(x, y) = \text{sqrt}((x_1 - y_1)^2 - (x_2 - y_2)^2)$)

2. Derive **Nearest Neighbors**

a. Calculate **distance** between newly entered input and other training data input values

b. Compile top ***K value*** (**nearest neighbors**) based on the **magnitude** of the distances

3. Predict **outputs** for testing dataset

Calculate outputs for **testing dataset** using the **K value** from the **training dataset** and the **max() function**

Methodology (LR)

1. Develop **Predicting Function**

Create **simple framework** of predicting function using **sigmoid function, weights,** and **biases**

2. Predict **coefficients**

a. Predict values for the coefficients of **weights** and **biases** using **random math function**

b. Check values for **costs/errors** using **stochastic gradient descent**

Costs present

Costs absent

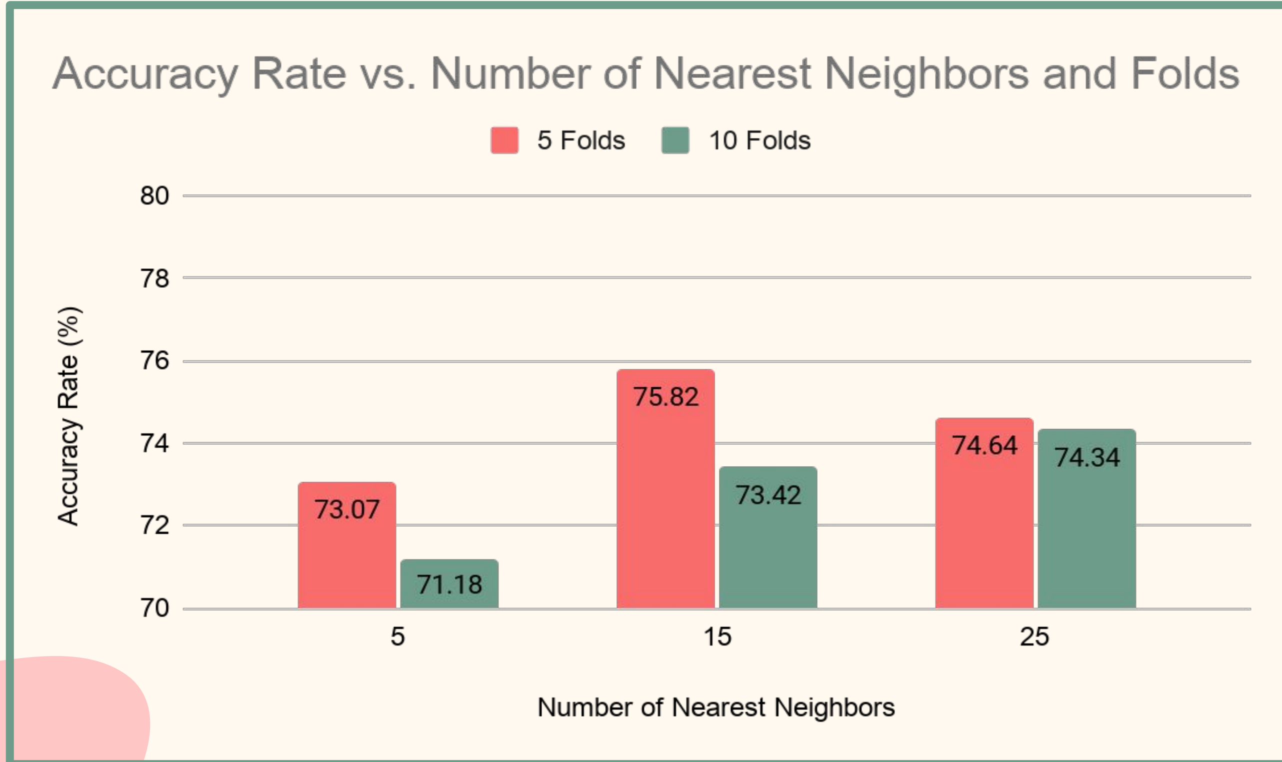
3. Predict **outputs** for testing dataset

Calculate outputs for **testing dataset** using the **coefficients** of weights and biases from the **training dataset**

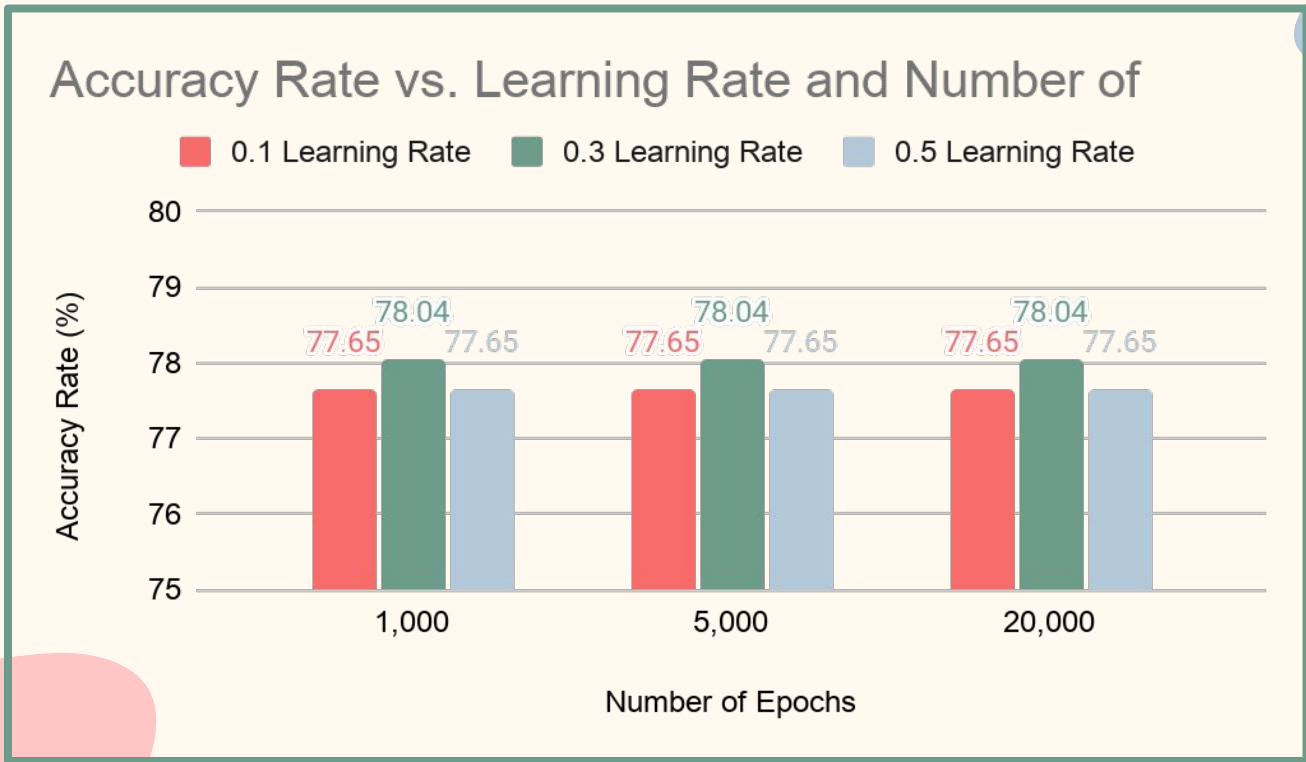
Methodology (DT)

1. Calculate **Gini Index**	2. Create a **split**	3. Build a **"Tree"**	4. Predict **outputs**
Calculate Gini Index based on **proportion** and **organization** of **groups** (ideal: **0**)	Split training dataset into groups **based on** calculated **Gini Index** and check **costs**	Create **nodes** other than the root ones and **recursively split** some of them for each data group	Predict outputs by **navigating the tree** through different nodes until the **final output** is produced for each input

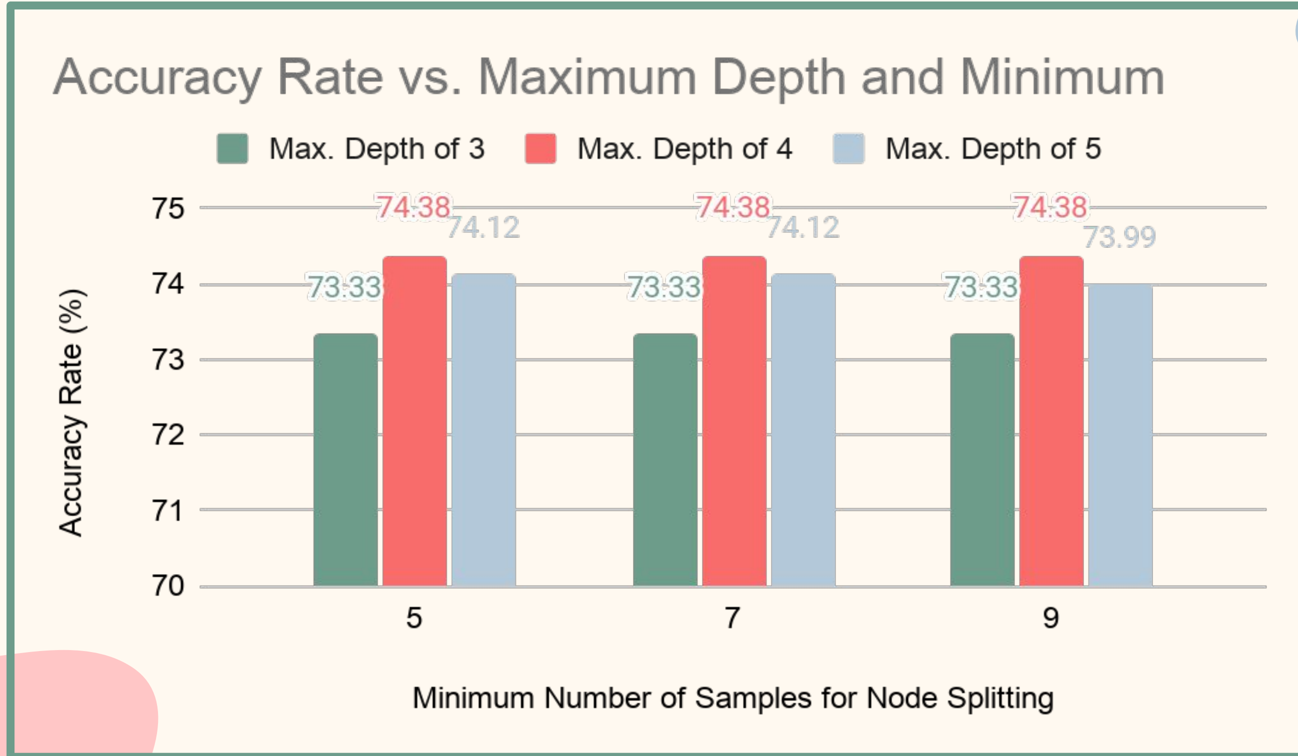
Results (Figure 1)



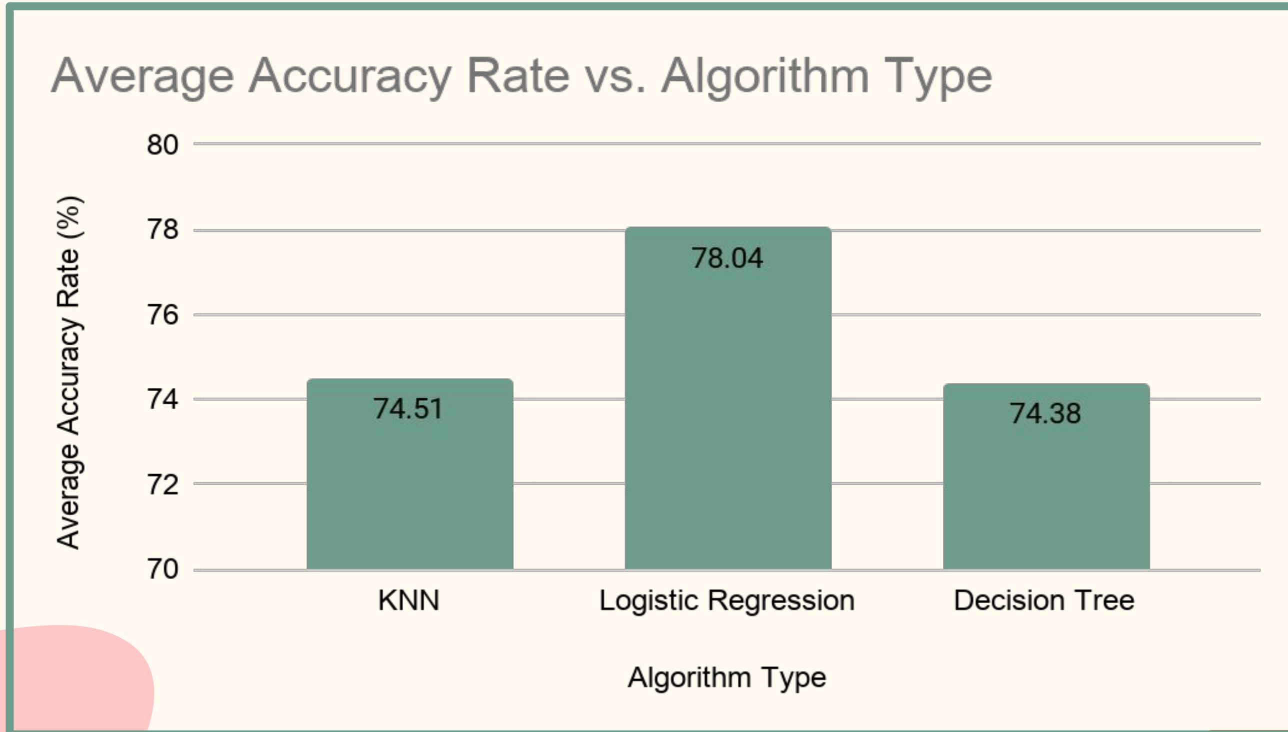
Results (Figure 2)



Results (Figure 3)



Results (Figure 4)



The Early Detection of Type 2 Diabetes Using Machine Learning Algorithms

Shridha Rajeswar
duPont Manual High School

Introduction

Type 2 Diabetes

Pancreas **doesn't produce** sufficient amounts of **insulin** to regulate glucose / body is **insulin-resistant → excess glucose in blood**

Credit: fundacionarlosslim.org

Context

380 million
people live with Type 2 Diabetes on a **daily basis**

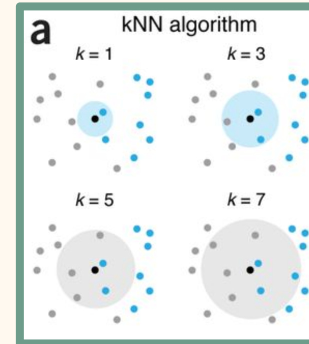
Invasive/Inaccurate
Current diagnosis equipment often produce **false results** (false "positives" and "negatives")

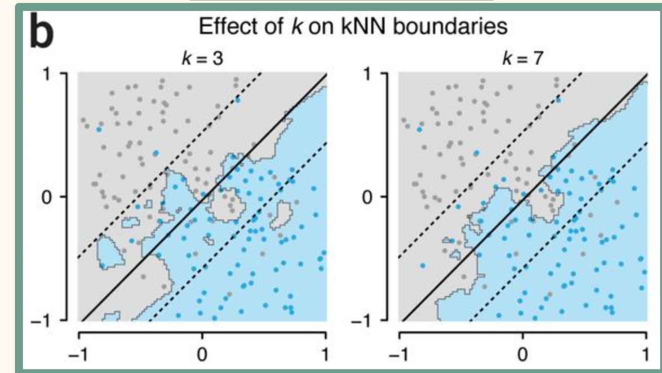
Machine Learning

- ML enables machines to **perform tasks** with **artificial intelligence**
- Multiple types for different purposes
  - **K-Nearest Neighbors (KNN)**
  - **Logistic Regression (LR)**
  - **Decision Tree (DT)**
- Input: diabetic parameter
- Output: diabetic condition

K-Nearest Neighbors

- Used for **classification** and **regression problems**
- Behaves like humans; **affected by** parameters that have **neighboring** or **close values**
  - Data **classified** into different categories **based on proximity** to one another
- Components - **folds** and **nearest neighbors**

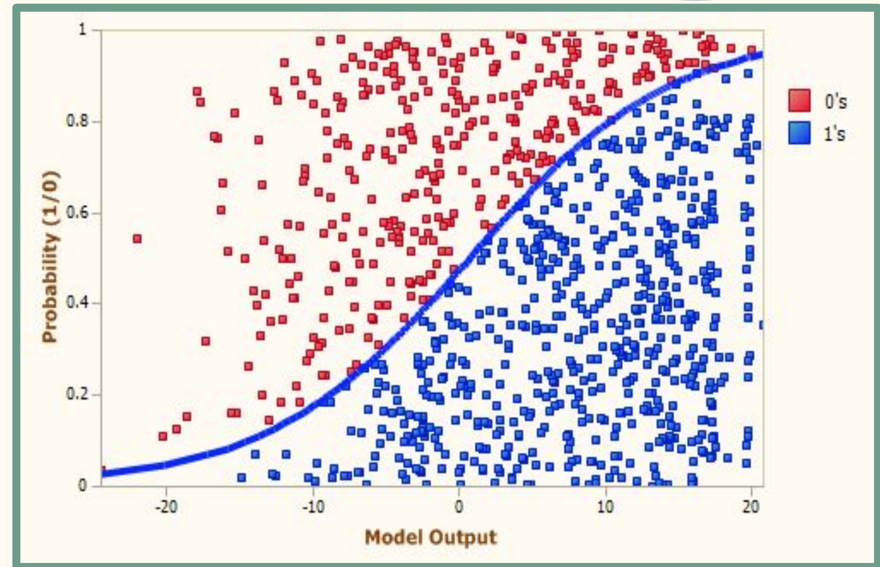




Credit: www.researchgate.net

Logistic Regression

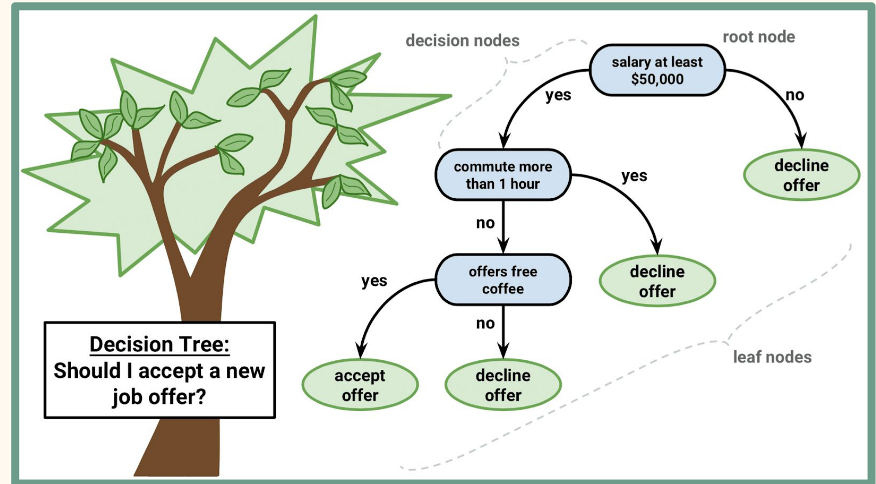
- Used for **regression problems**
- Uses **sigmoid** and **similar functions** to the ones in Artificial Neural Networks; works in **two different phases** (training and testing)
  - **Multiple functions** carried out **to produce output value** from input values
- Components - **learning rate** and **iterations/epochs**



Credit: fusionanalysticsworld.com

Decision Tree

- Used for **classification** and **regression problems**
- Behaves like a tree; continually **splits nodes** at different levels **to categorize input** and **branches off** when close to **predicting outputs**
  - **Gini index** determines **quality of split** to ensure that outputs are accurate
- Components - **maximum depth** and **minimum number of samples for node splitting**



Credit: mediumcom

Engineering Goal

Create **three** different types of ML algorithms that **successfully** predict binary outputs indicating a patient's diabetic condition based on given input values

- Independent variables: **differs** for each ML algorithm type
- Dependent variables: **accuracy rates**
- Constant variables: **folds** (excluding KNN)

KNN

folds, neighbors

LR

learning rate, iterations

DT

max. depth, min. samples for node splitting

Methodology (KNN)

1. Calculate **Euclidean Distance**

Calculate **distance** between two rows in training dataset using ***euclidean_distance()*** **function** ($d(x, y) = sqrt((x_1 - y_1)^2 - (x_2 - y_2)^2)$)

2. Derive **Nearest Neighbors**

a. Calculate **distance** between newly entered input and other training data input values

b. Compile top ***K value*** (**nearest neighbors**) based on the **magnitude** of the distances

3. Predict **outputs** for testing dataset

Calculate outputs for **testing dataset** using the **K value** from the **training dataset** and the **max() function**

Methodology (LR)

1. Develop **Predicting Function**

Create **simple framework** of predicting function using **sigmoid function, weights,** and **biases**

2. Predict **coefficients**

a. Predict values for the coefficients of **weights** and **biases** using **random math function**

b. Check values for **costs/errors** using **stochastic gradient descent**

Costs present (back to a)

Costs absent (on to step 3)

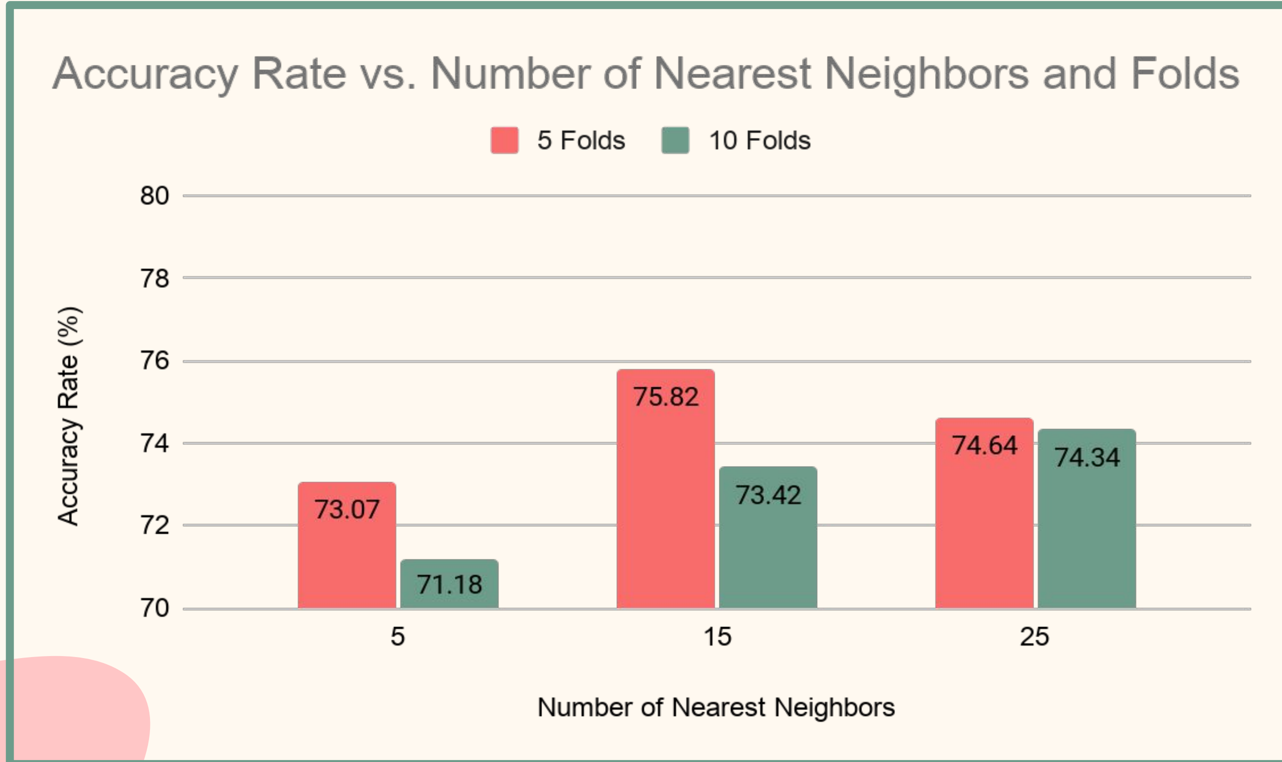
3. Predict **outputs** for testing dataset

Calculate outputs for **testing dataset** using the **coefficients** of weights and biases from the **training dataset**

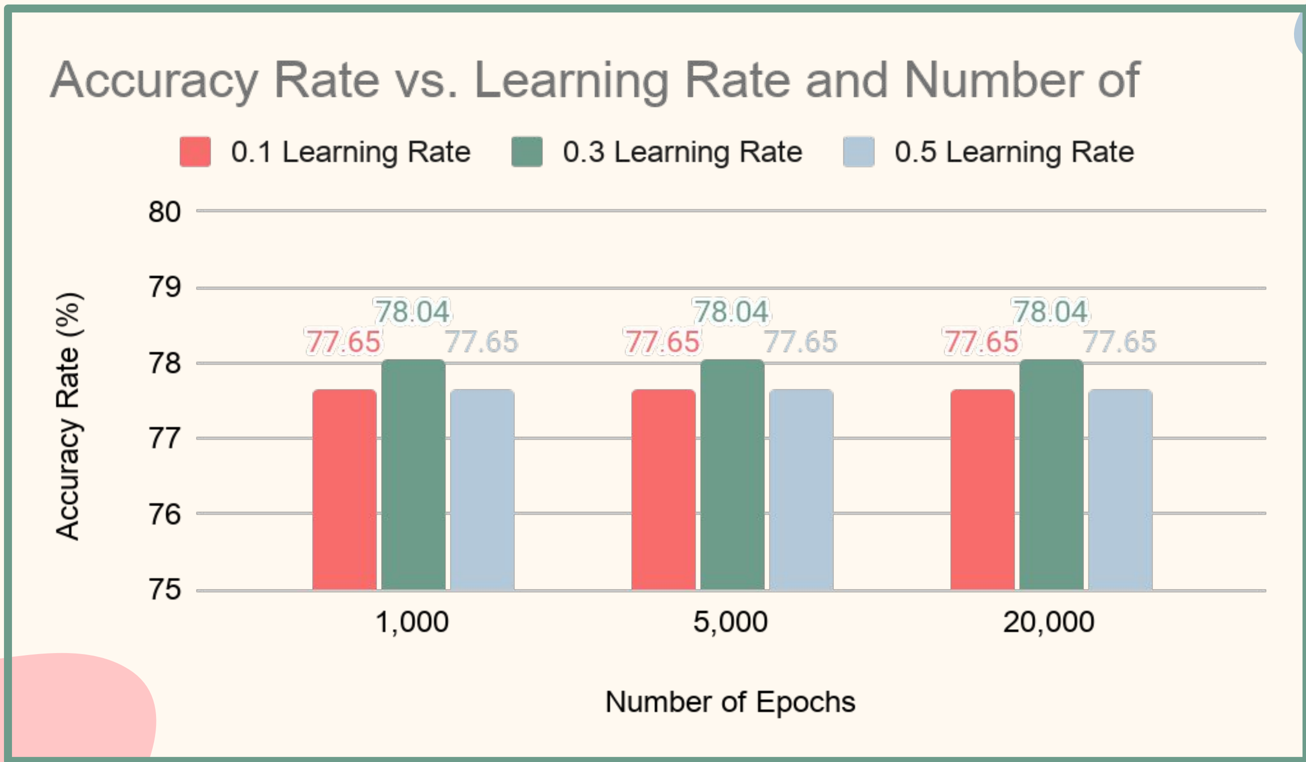
Methodology (DT)

1. Calculate **Gini Index**	2. Create a **split**	3. Build a **"Tree"**	4. Predict **outputs**
Calculate Gini Index based on **proportion** and **organization** of **groups** (ideal: **0**)	Split training dataset into groups **based on** calculated **Gini Index** and check **costs**	Create **nodes** other than the root ones and **recursively split** some of them for each data group	Predict outputs by **navigating the tree** through different nodes until the **final output** is produced for each input

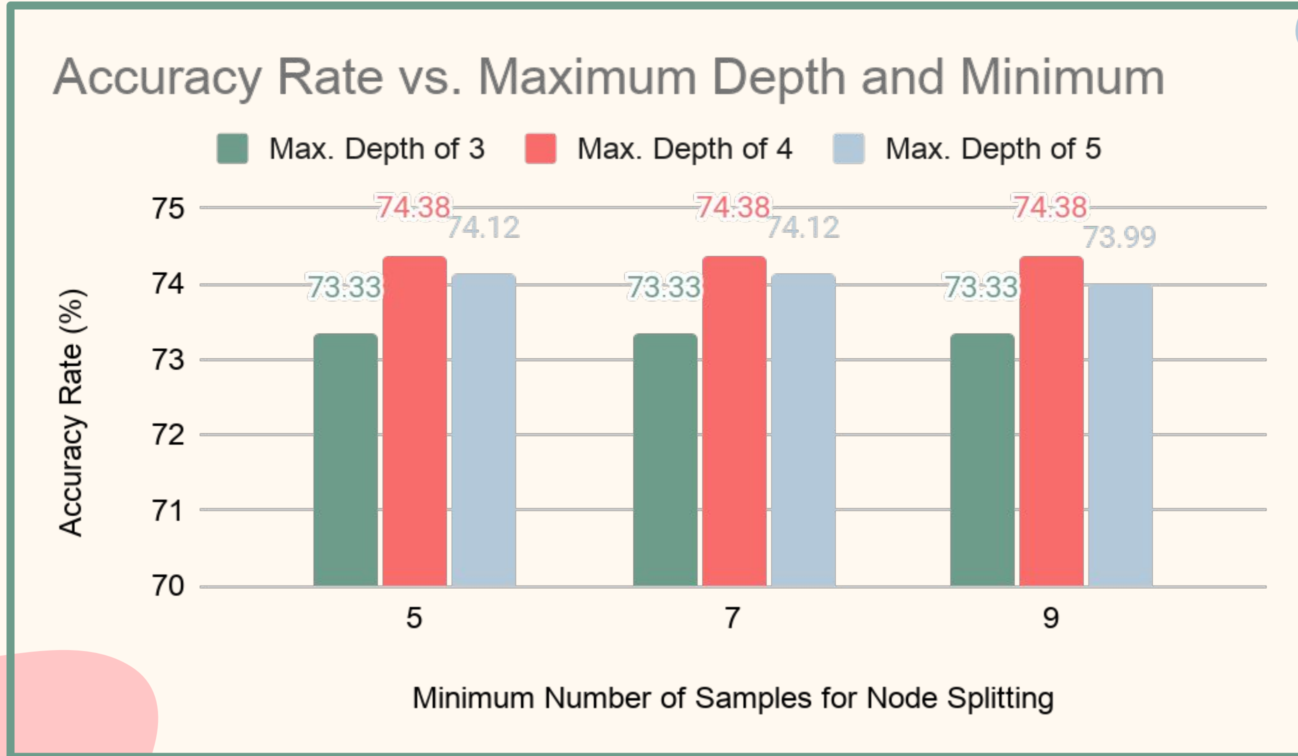
Results (Figure 1)



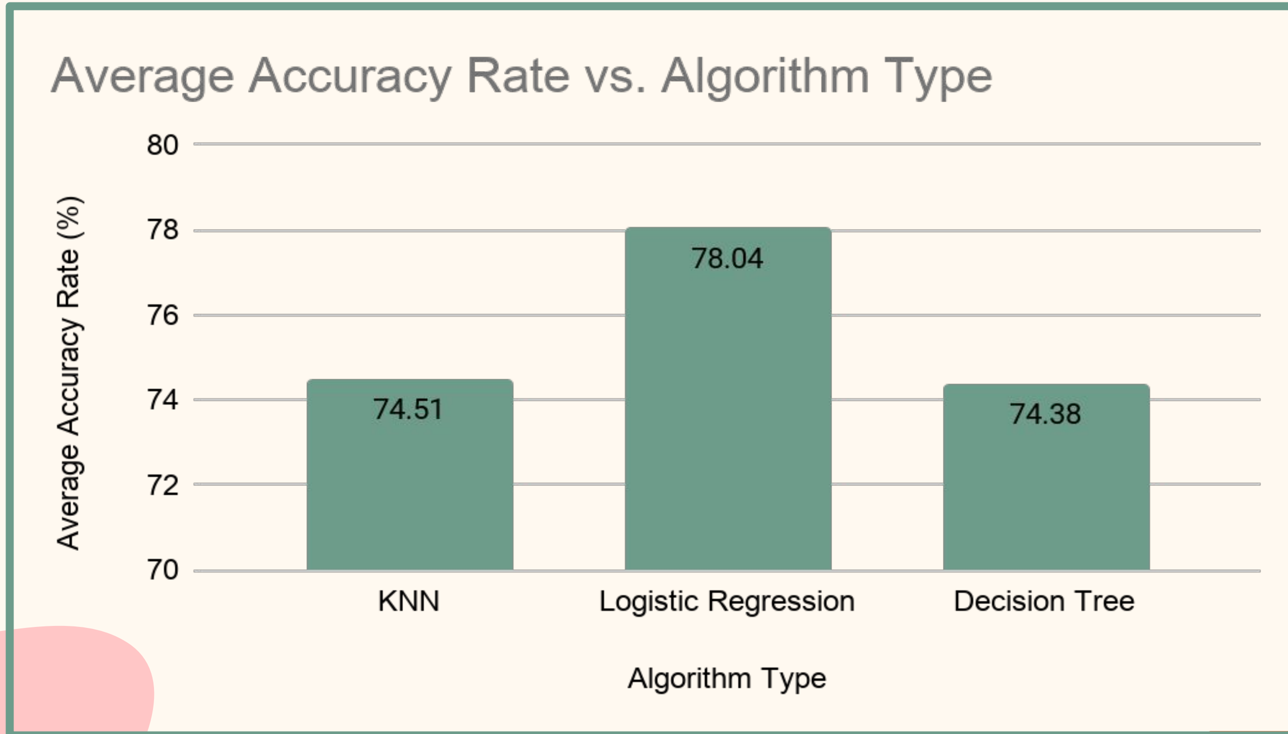
Results (Figure 2)



Results (Figure 3)



Results (Figure 4)



Trends

Logistic Regression was the most accurate of all the programs overall (**78.04%**)

KNN: Most accurate experimental group had **5 folds**; most accurate program had **15 nearest neighbors** (**75.82%**)

LR: Most accurate experimental group had **0.3 learning rate**; most accurate program had **same performance** (**78.04%**)

DT: Most accurate experimental group had **max. depth of 4**; most accurate program had **same performance** (**74.38%**)

Data Analysis

Comparison (1 vs. 2)	Mean 1	Mean 2	Variance 1	Variance 2	T-value (observed)	T-value (0.005 level)	Hypothesis
5 folds vs 10 folds (KNN)	74.51	72.98	1.903	2.642	1.243	9.925	Null accepted
0.3 learning rate vs. 5 folds	78.04	74.51	0	1.903	4.432	9.925	Null accepted
5 folds vs maximum depth of 4	74.51	74.38	1.903	0	0.1632	9.925	Null accepted
0.3 learning rate vs maximum depth of 4	78.04	74.38	0	0			Inconclusive (no variation)

Figure 5

- **Multiple paired t-tests** were conducted for statistical significance
  - All alternate hypotheses were rejected; the **null hypotheses were accepted**
  - Test with highest average accuracy rates of **LR and KNN algorithms** had **most significance (4.432)**
  - T-value was **undefined** for **LR and DT algorithms** as both had **no variation** in accuracy rates when changing their respective variables

Conclusion

Goal achieved

All the three ML algorithms successfully predicted outputs

Ideal KNN programs

Middle range of nearest neighbors, **less** folds

Ideal LT programs

Middle range of learning rate, iterations have **no effect**

Ideal DT programs

Middle range of max. depth, min. samples for node splitting has no effect

Future Recommendations

Hardware Implementation

Machine Learning algorithms could be implemented **in sensory detection hardware prototypes** to test **practicality** and other forms of **effectiveness**

Testing other ML Algorithms

More types of Machine Learning algorithms (**Naive Bayes, Support Vector Machines, etc.**) and independent variables could be tested to **improve accuracy rates**

Acknowledgements

I would like to acknowledge my teacher, **Ms. Kathy Fries** and the website Machine Learning Mastery by **Mr. Jason Brownlee**. Ms. Fries helped me in researching a topic that was both enriching and suitable for me. The website by Mr. Brownlee had many useful resources for coding ML algorithms in Python, and some of the complex machine learning concepts were broken down into simpler and more understable information.

Visuals of Code (KNN)

```
# Load a CSV file
def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
		print('[%s] => %d' % (value, i))
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup
```

```
# Find the min and max values for each column
def dataset_minmax(dataset):
	minmax = list()
	for i in range(len(dataset[0])):
		col_values = [row[i] for row in dataset]
		value_min = min(col_values)
		value_max = max(col_values)
		minmax.append([value_min, value_max])
	return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
	for row in dataset:
		for i in range(len(row)):
			row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
	distance = 0.0
	for i in range(len(row1)-1):
		distance += (row1[i] - row2[i])**2
	return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
	distances = list()
	for train_row in train:
		dist = euclidean_distance(test_row, train_row)
		distances.append((train_row, dist))
	distances.sort(key=lambda tup: tup[1])
	neighbors = list()
	for i in range(num_neighbors):
		neighbors.append(distances[i][0])
	return neighbors
```

Visuals of Code (KNN)

```
# Make a prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
        neighbors = get_neighbors(train, test_row, num_neighbors)
        output_values = [row[-1] for row in neighbors]
        prediction = max(set(output_values), key=output_values.count)
        return prediction

# Make a prediction with KNN on Diabetes Dataset
filename = 'data.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
        str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# define model parameter
num_neighbors = 5
# define a new record
row = [120,74,10,0,64.9,0.15,35]
# predict the label
label = predict_classification(dataset, row, num_neighbors)
print('Data=%s, Predicted: %s' % (row, label))
```

Visuals of Code (LR)

```
# Make a prediction with coefficients
def predict(row, coefficients):
        yhat = coefficients[0]
        for i in range(len(row)-1):
                yhat += coefficients[i + 1] * row[i]
        return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
        coef = [0.0 for i in range(len(train[0]))]
        for epoch in range(n_epoch):
                sum_error = 0
                for row in train:
                        yhat = predict(row, coef)
                        error = row[-1] - yhat
                        sum_error += error**2
                        coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
                        for i in range(len(row)-1):
                                coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
                print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
        return coef
```

Visuals of Code (LR)

```
# Make a prediction
from math import exp

# Make a prediction with coefficients
def predict(row, coefficients):
	yhat = coefficients[0]
	for i in range(len(row)-1):
		yhat += coefficients[i + 1] * row[i]
	return 1.0 / (1.0 + exp(-yhat))

# test predictions
dataset = [
[2.7810836,2.550537003,0],
	[1.465489372,2.362125076,0],
	[3.396561688,4.400293529,0],
	[1.38807019,1.850220317,0],
	[3.06407232,3.005305973,0],
	[7.627531214,2.759262235,1],
	[5.332441248,2.088626775,1]
]
coef = [-1.5495305815023432, 2.6929943470390043, -3.9818757514207848]
for row in dataset:
	yhat = predict(row, coef)
	print("Expected=%.3f, Predicted=%.3f [%d]" % (row[-1], yhat, round(yhat)))
```

Visuals of Code (DT)

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
	left, right = list(), list()
	for row in dataset:
		if row[index] < value:
			left.append(row)
		else:
			right.append(row)
	return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
	# count all samples at split point
	n_instances = float(sum([len(group) for group in groups]))
	# sum weighted Gini index for each group
	gini = 0.0
	for group in groups:
		size = float(len(group))
		# avoid divide by zero
		if size == 0:
			continue
		score = 0.0
		# score the group based on the score for each class
		for class_val in classes:
			p = [row[-1] for row in group].count(class_val) / size
			score += p * p
		# weight the group score by its relative size
		gini += (1.0 - score) * (size / n_instances)
	return gini
```

Visuals of Code (DT)

```
# Select the best split point for a dataset
def get_split(dataset):
	class_values = list(set(row[-1] for row in dataset))
	b_index, b_value, b_score, b_groups = 999, 999, 999, None
	for index in range(len(dataset[0])-1):
		for row in dataset:
			groups = test_split(index, row[index], dataset)
			gini = gini_index(groups, class_values)
			if gini < b_score:
				b_index, b_value, b_score, b_groups = index, row[index], gini, groups
	return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
	outcomes = [row[-1] for row in group]
	return max(set(outcomes), key=outcomes.count)
```

Visuals of Code (DT)

```
# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
	left, right = node['groups']
	del(node['groups'])
	# check for a no split
	if not left or not right:
		node['left'] = node['right'] = to_terminal(left + right)
		return
	# check for max depth
	if depth >= max_depth:
		node['left'], node['right'] = to_terminal(left), to_terminal(right)
		return
	# process left child
	if len(left) <= min_size:
		node['left'] = to_terminal(left)
	else:
		node['left'] = get_split(left)
		split(node['left'], max_depth, min_size, depth+1)
	# process right child
	if len(right) <= min_size:
		node['right'] = to_terminal(right)
	else:
		node['right'] = get_split(right)
		split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
	root = get_split(train)
	split(root, max_depth, min_size, 1)
	return root
```

Visuals of Code (DT)

```
# Print a decision tree
def print_tree(node, depth=0):
        if isinstance(node, dict):
                print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
                print_tree(node['left'], depth+1)
                print_tree(node['right'], depth+1)
        else:
                print('%s[%s]' % ((depth*' ', node)))
```

Visuals of Code (DT)

```
# Print a decision tree
def print_tree(node, depth=0):
	if isinstance(node, dict):
		print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
		print_tree(node['left'], depth+1)
		print_tree(node['right'], depth+1)
	else:
		print('%s[%s]' % ((depth*' ', node)))
```

```
# Make a prediction with a decision tree
def predict(node, row):
	if row[node['index']] < node['value']:
		if isinstance(node['left'], dict):
			return predict(node['left'], row)
		else:
			return node['left']
	else:
		if isinstance(node['right'], dict):
			return predict(node['right'], row)
		else:
			return node['right']
```

Trends

Logistic Regression was the most accurate of all the programs overall (**78.04%**)

KNN: Most accurate experimental group had **5 folds**; most accurate program had **15 nearest neighbors** (**75.82%**)

LR: Most accurate experimental group had **0.3 learning rate**; most accurate program had **same performance** (**78.04%**)

DT: Most accurate experimental group had **max. depth of 4**; most accurate program had **same performance** (**74.38%**)

Data Analysis

Comparison (1 vs. 2)	Mean 1	Mean 2	Variance 1	Variance 2	T-value (observed)	T-value (0.005 level)	Hypothesis
5 folds vs 10 folds (KNN)	74.51	72.98	1.903	2.642	1.243	9.925	Null accepted
0.3 learning rate vs. 5 folds	78.04	74.51	0	1.903	4.432	9.925	Null accepted
5 folds vs maximum depth of 4	74.51	74.38	1.903	0	0.1632	9.925	Null accepted
0.3 learning rate vs maximum depth of 4	78.04	74.38	0	0			Inconclusive (no variation)

Figure 5

- **Multiple paired t-tests** were conducted for statistical significance
  - All alternate hypotheses were rejected; the **null hypotheses were accepted**
  - Test with highest average accuracy rates of **LR and KNN algorithms** had **most significance (4.432)**
  - T-value was **undefined** for **LR and DT algorithms** as both had **no variation** in accuracy rates when changing their respective variables

Conclusion

Goal achieved

All the three ML algorithms successfully predicted outputs

Ideal KNN programs

Middle range of nearest neighbors, **less** folds

Ideal LT programs

Middle range of learning rate, iterations have **no effect**

Ideal DT programs

Middle range of max. depth, min. samples for node splitting has no effect

Future Recommendations

Hardware Implementation

Machine Learning algorithms could be implemented **in sensory detection hardware prototypes** to test **practicality** and other forms of **effectiveness**

Testing other ML Algorithms

More types of Machine Learning algorithms (**Naive Bayes, Support Vector Machines, etc.**) and independent variables could be tested to **improve accuracy rates**

Acknowledgements

I would like to acknowledge my teacher, **Ms. Kathy Fries** and the website Machine Learning Mastery by **Mr. Jason Brownlee**. Ms. Fries helped me in researching a topic that was both enriching and suitable for me. The website by Mr. Brownlee had many useful resources for coding ML algorithms in Python, and some of the complex machine learning concepts were broken down into simpler and more understable information.

Visuals of Code (KNN)

```
# Load a CSV file
def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
		print('[%s] => %d' % (value, i))
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup
```

```
# Find the min and max values for each column
def dataset_minmax(dataset):
	minmax = list()
	for i in range(len(dataset[0])):
		col_values = [row[i] for row in dataset]
		value_min = min(col_values)
		value_max = max(col_values)
		minmax.append([value_min, value_max])
	return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
	for row in dataset:
		for i in range(len(row)):
			row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
	distance = 0.0
	for i in range(len(row1)-1):
		distance += (row1[i] - row2[i])**2
	return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
	distances = list()
	for train_row in train:
		dist = euclidean_distance(test_row, train_row)
		distances.append((train_row, dist))
	distances.sort(key=lambda tup: tup[1])
	neighbors = list()
	for i in range(num_neighbors):
		neighbors.append(distances[i][0])
	return neighbors
```

Visuals of Code (KNN)

```
# Make a prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
	neighbors = get_neighbors(train, test_row, num_neighbors)
	output_values = [row[-1] for row in neighbors]
	prediction = max(set(output_values), key=output_values.count)
	return prediction

# Make a prediction with KNN on Diabetes Dataset
filename = 'data.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
	str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# define model parameter
num_neighbors = 5
# define a new record
row = [120,74,10,0,64.9,0.15,35]
# predict the label
label = predict_classification(dataset, row, num_neighbors)
print('Data=%s, Predicted: %s' % (row, label))
```

Visuals of Code (LR)

```
# Make a prediction with coefficients
def predict(row, coefficients):
        yhat = coefficients[0]
        for i in range(len(row)-1):
                yhat += coefficients[i + 1] * row[i]
        return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients using stochastic gradient descent
def coefficients_sgd(train, l_rate, n_epoch):
        coef = [0.0 for i in range(len(train[0]))]
        for epoch in range(n_epoch):
                sum_error = 0
                for row in train:
                        yhat = predict(row, coef)
                        error = row[-1] - yhat
                        sum_error += error**2
                        coef[0] = coef[0] + l_rate * error * yhat * (1.0 - yhat)
                        for i in range(len(row)-1):
                                coef[i + 1] = coef[i + 1] + l_rate * error * yhat * (1.0 - yhat) * row[i]
                print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
        return coef
```

Visuals of Code (LR)

```
# Make a prediction
from math import exp

# Make a prediction with coefficients
def predict(row, coefficients):
	yhat = coefficients[0]
	for i in range(len(row)-1):
		yhat += coefficients[i + 1] * row[i]
	return 1.0 / (1.0 + exp(-yhat))

# test predictions
dataset = [
[2.7810836,2.550537003,0],
	[1.465489372,2.362125076,0],
	[3.396561688,4.400293529,0],
	[1.38807019,1.850220317,0],
	[3.06407232,3.005305973,0],
	[7.627531214,2.759262235,1],
	[5.332441248,2.088626775,1]
]
coef = [-1.5495305815023432, 2.6929943470390043, -3.9818757514207848]
for row in dataset:
	yhat = predict(row, coef)
	print("Expected=%.3f, Predicted=%.3f [%d]" % (row[-1], yhat, round(yhat)))
```

Visuals of Code (DT)

```
# Split a dataset based on an attribute and an attribute value
def test_split(index, value, dataset):
	left, right = list(), list()
	for row in dataset:
		if row[index] < value:
			left.append(row)
		else:
			right.append(row)
	return left, right

# Calculate the Gini index for a split dataset
def gini_index(groups, classes):
	# count all samples at split point
	n_instances = float(sum([len(group) for group in groups]))
	# sum weighted Gini index for each group
	gini = 0.0
	for group in groups:
		size = float(len(group))
		# avoid divide by zero
		if size == 0:
			continue
		score = 0.0
		# score the group based on the score for each class
		for class_val in classes:
			p = [row[-1] for row in group].count(class_val) / size
			score += p * p
		# weight the group score by its relative size
		gini += (1.0 - score) * (size / n_instances)
	return gini
```

Visuals of Code (DT)

```
# Select the best split point for a dataset
def get_split(dataset):
	class_values = list(set(row[-1] for row in dataset))
	b_index, b_value, b_score, b_groups = 999, 999, 999, None
	for index in range(len(dataset[0])-1):
		for row in dataset:
			groups = test_split(index, row[index], dataset)
			gini = gini_index(groups, class_values)
			if gini < b_score:
				b_index, b_value, b_score, b_groups = index, row[index], gini, groups
	return {'index':b_index, 'value':b_value, 'groups':b_groups}

# Create a terminal node value
def to_terminal(group):
	outcomes = [row[-1] for row in group]
	return max(set(outcomes), key=outcomes.count)
```

Visuals of Code (DT)

```
# Create child splits for a node or make terminal
def split(node, max_depth, min_size, depth):
	left, right = node['groups']
	del(node['groups'])
	# check for a no split
	if not left or not right:
		node['left'] = node['right'] = to_terminal(left + right)
		return
	# check for max depth
	if depth >= max_depth:
		node['left'], node['right'] = to_terminal(left), to_terminal(right)
		return
	# process left child
	if len(left) <= min_size:
		node['left'] = to_terminal(left)
	else:
		node['left'] = get_split(left)
		split(node['left'], max_depth, min_size, depth+1)
	# process right child
	if len(right) <= min_size:
		node['right'] = to_terminal(right)
	else:
		node['right'] = get_split(right)
		split(node['right'], max_depth, min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
	root = get_split(train)
	split(root, max_depth, min_size, 1)
	return root
```

Visuals of Code (DT)

```
# Print a decision tree
def print_tree(node, depth=0):
        if isinstance(node, dict):
                print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
                print_tree(node['left'], depth+1)
                print_tree(node['right'], depth+1)
        else:
                print('%s[%s]' % ((depth*' ', node)))
```

Visuals of Code (DT)

```
# Print a decision tree
def print_tree(node, depth=0):
	if isinstance(node, dict):
		print('%s[X%d < %.3f]' % ((depth*' ', (node['index']+1), node['value'])))
		print_tree(node['left'], depth+1)
		print_tree(node['right'], depth+1)
	else:
		print('%s[%s]' % ((depth*' ', node)))
```

```
# Make a prediction with a decision tree
def predict(node, row):
	if row[node['index']] < node['value']:
		if isinstance(node['left'], dict):
			return predict(node['left'], row)
		else:
			return node['left']
	else:
		if isinstance(node['right'], dict):
			return predict(node['right'], row)
		else:
			return node['right']
```