COMP7703 – Final Exam Submission

Joel Thomas 44793203

Introduction:

The purpose of this informal report is to analyse the provided dataset using exploratory data analysis (EDA) and machine learning techniques as part of the final exam submission for COMP7703. Python has been used in entirety for all of the programming-related tasks and all of the code (Python $.py$ files and a Jupyter Notebook $.ipynb$ file) has been attached for reference at the end of the report.

Preprocessing of the Food Nutrients dataset:

Initially, we observe that the dataset is provided as CSV format, so the dataset is loaded into a Pandas Dataframe (Pandas library). It can be identified that there are:

- $N = 1535$ samples or rows
- $d = 252$ features or columns
    - Number of categorical features $= 2$ ("Public Food Key" and "Food Name")
    - Number of numerical features $= d - 2 = 250$

	Public Food Key	Classification	Food Name	Energy, with dietary fibre	Energy, without dietary fibre	Moisture (water)	Protein	Nitrogen	Total Fat	Ash	...	Proline	Unnamed: 243	Serine	Unnamed: 245	Threonine
0	NaN	NaN	NaN	kJ	kJ	g	g	g	g	g	...	mg/g N	mg	mg/g N	mg	mg/g N
1	F009117	11101.0	Tea, green, plain, without milk	7	7	95.7	0.2	0.04	0.1	0	...	NaN	NaN	NaN	NaN	NaN
2	F009125	11101.0	Tea, regular, black, brewed from leaf or teaba...	2	2	99.8	0.1	0.02	0	0	...	NaN	NaN	NaN	NaN	NaN
3	F009115	11103.0	Tea, decaffeinated, black, brewed from leaf or...	6	6	99.8	0.1	0.02	0.1	0	...	NaN	NaN	NaN	NaN	NaN
4	F003017	11201.0	Coffee, black, from instant coffee powder	5	4	99.2	0.2	0.03	0	0.1	...	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1530	F004196	31502.0	Gelatine, all types	1449	1449	11	84.4	15.2	0.4	1.8	...	821	12479	252	3830	137
1531	F005647	32102.0	Milk, human/breast, mature, fluid	286	286	87.5	1.3	0.2	4.2	0.2	...	NaN	NaN	NaN	NaN	NaN
1532	F003301	34101.0	Crocodile, tail fillet, raw	442	442	76.2	22.5	3.6	1.6	1	...	NaN	NaN	NaN	NaN	NaN
1533	F003299	34101.0	Crocodile, back leg, raw	455	455	76.1	22	3.52	2.2	0.9	...	NaN	NaN	NaN	NaN	NaN
1534	F003300	34101.0	Crocodile, cooked, no added fat	748	748	60.2	37.1	5.93	3.2	1.6	...	NaN	NaN	NaN	NaN	NaN

1535 rows × 252 columns

Figure 1: Preview of the Food Nutrients dataset

We first randomly shuffle the entire dataset as there appears to be an innate ordering present – for example, the teas appear together, then the coffees appear together, etc. Upon initial inspection of figure 1, it is seen that there are several features that contain NaN values so rather than removing every sample containing a NaN value (which would result in a massive reduction of the number of samples in the data), we choose to remove all features that contain NaN values instead. We also remove the first row since it is trivial and only contains the unit of

measurement for each feature. After this step, we have successfully reduced the number of features from $d = 252$ to $d = d' = 60$.

The most interesting logical feature that forms the basis of our analysis as a target for our intended machine learning technique is the "Food Name" feature which represents the explicit full food name for each stored sample. Here, we observe that the first word for every sample represents the major food type/group – for example, tea, coffee, bread, beef, etc. A quick scan of the samples in Excel representing different meats indicates that there are several more samples for the meats compared to others. Hence, for the purpose of our discussion, we choose to narrow down our focus to solely on the four most popular meats – beef, chicken, lamb and pork. A viable problem here is use machine learning to help predict and classify which meat a given sample belongs to using the remaining $d - 1 = 59$ columns as viable features to potentially feed as inputs into our machine learning model.

Before proceeding any further, we first convert the only remaining categorical feature (our target) "Food Name" into numerical features by discretising them – e.g. we use 0 to represent beef, 1 to represent chicken, 2 to represent lamb and 3 to represent pork. This form of discretisation is based on the number of unique categories encountered within a categorical feature.

	Energy, with dietary fibre	Energy, without dietary fibre	Moisture (water)	Protein	Nitrogen	Total Fat	Ash	Total dietary fibre	Alcohol	Total sugars	...	Unnamed: 193	Unnamed: 199	Unnamed: 203	Unnamed: 207	Unnamed: 209	Unname 21
0	738	738	66.9	22.8	3.66	9.5	1.2	0.0	0	0.0	...	13.82	17.27	2.68	0.35	33.779	448.2
1	472	470	72.6	19.4	3.10	1.2	3.0	0.2	0	0.2	...	1.18	2.36	1.77	0.21	5.316	7.0
2	953	953	58.4	28.7	4.60	12.6	1.1	0.0	0	0.0	...	67.23	91.26	23.20	1.10	181.695	740.3
3	573	573	71.1	21.7	3.47	5.5	1.2	0.0	0	0.0	...	18.88	29.65	3.49	0.34	52.017	242.4
4	528	528	72.7	21.9	3.50	4.2	1.1	0.0	0	0.0	...	18.97	36.05	9.77	0.48	64.787	167.8
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
333	602	602	71.6	20.4	3.26	6.9	1.0	0.0	0	0.0	...	32.32	61.42	16.65	0.82	110.386	285.9
334	910	910	57.5	29.9	4.79	10.9	1.3	0.0	0	0.0	...	35.34	54.56	7.33	0.62	97.231	485.3
335	848	848	64.1	21.2	3.39	13.2	1.1	0.0	0	0.0	...	25.00	27.93	4.37	0.49	57.294	624.1
336	1961	1961	38.4	16.3	2.61	45.5	0.9	0.0	0	0.0	...	0.00	43.36	43.36	6.03	86.724	216.8
337	480	480	72.8	23.3	3.73	2.3	1.2	0.0	0	0.0	...	24.95	33.13	5.18	0.28	63.258	82.3

338 rows × 60 columns

Figure 2: Preview of the fully numerical dataset obtained after discretising categorical features, last column (60, not shown) is "Food Name"

Performing PCA to reduce dimensionality:

With $d = 59$ features (excluding target), this represents a high-dimensional problem. We first conduct Principal Component Analysis (PCA) to reduce the dimensionality of the dataset by means of automated feature extraction. Let X represent the entire input feature set, W represent the matrix of eigenvectors for the sample covariance matrix of X ordered according to corresponding descending eigenvalues (largest to smallest) and Z represent the linear projection of $X - \bar{X}$ (centred) on the direction of W. Observe that these are the matrix rather than vector representations of the formula for performing PCA (usually one sample rather than entire dataset) but the output is still correct regardless. Thus, mathematically, we have:

$$\dim(X) = 338 \times 59, \qquad \dim(W) = 59 \times 59, \qquad Z = (W^T(X - \bar{X})^T)^T \rightarrow \dim(Z) = 338 \times 59$$

The additional transposes appearing in the formula are to make the dimensions match up for matrix multiplication. After conducting PCA, the proportion of variance explained by only the first three principal components is approximately $\frac{\lambda_1+\lambda_2+\lambda_3}{\lambda_1+\cdots+\lambda_{59}} \approx 99.48\%$.

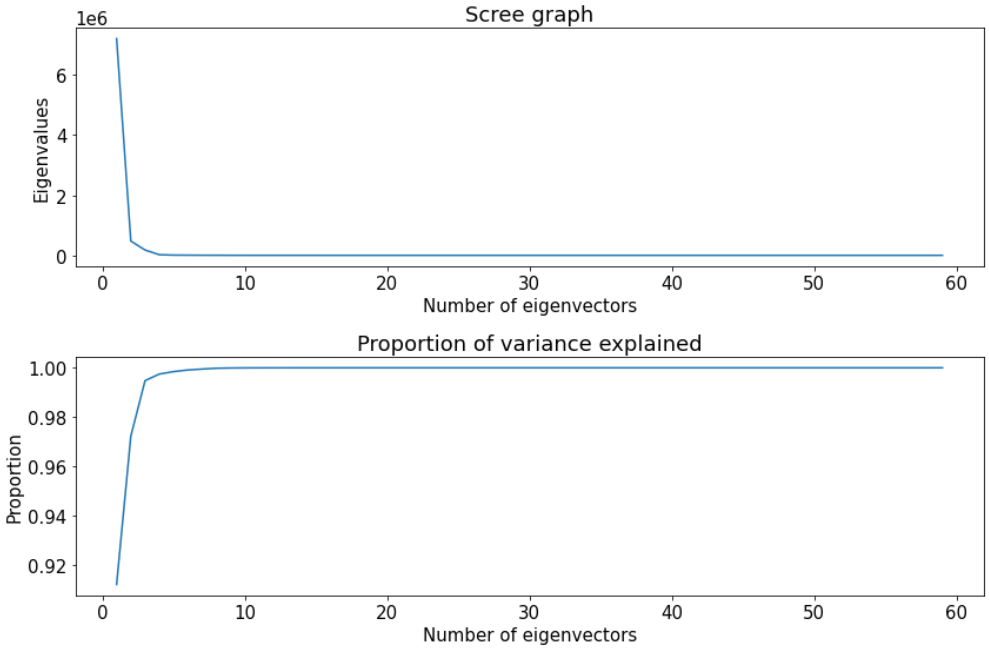


Figure 3: Scree graph and Proportion of variance explained graph after conducting PCA

From figure 3, observing the "elbow" of the proportion of variance explained graph, we choose to use only the first $k = 3$ principal components in training our machine learning model.

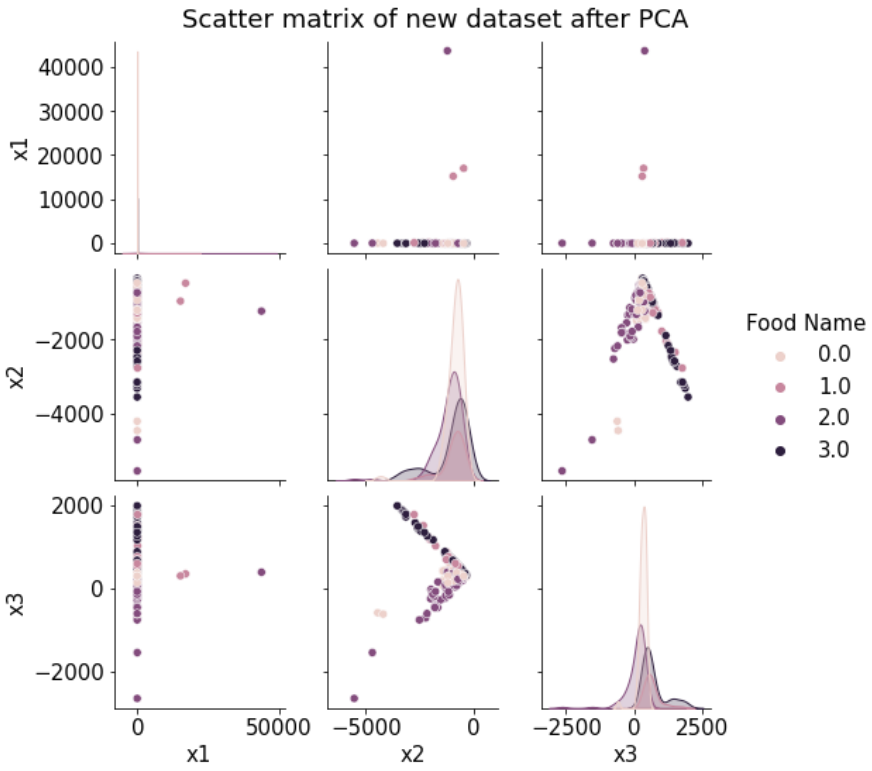


Figure 4: Scatterplot matrix/pair plot of the new dataset after conducting PCA. Recall we have used 0 to represent beef, 1 for chicken, 2 for lamb and 3 for pork.

From figure 4, the cross scatter plots for x_1 and x_2, x_1 and x_3 and x_2 and x_3 do not represent an almost clear separation of these projected features based on the underlying meat for each sample. In the next section, we will combine neural networks with deep learning via the TensorFlow library to generate highly non-linear boundaries for these scatter plots to help predict and classify a meat for a given sample.

Training an ensemble of Neural Networks:

One of the most fundamental theorems underlying probability theory is the Law of Large Numbers which states that the average of the results obtained from a large sample (size n) of experiments tends to the expected value of the experiment as more and more trials are performed ($n \to \infty$). Without getting into this theorem formally, in terms of machine learning, we can interpret it as evaluating the test set performance an ensemble of learners is better than relying on a single base learner. This is because, based on the run, we might have a highly biased test set performance – it might either be surprisingly too good to be true or quite poor on the contrary. An ensemble of learners helps generalise better to when the model encounters unseen data. A key condition for the usefulness of this method here is that each base learner is unstable (stochastic) in nature.

In this section, we shall train 9 individual neural networks (base learners) and then form an ensemble based on the trained models and finally compare their generalisation (hidden test set) performance later on using their average prediction. This method of combining learners is well known formally as Bagging. Recall that unstable (stochastic) algorithms such as neural networks profit from bagging because bootstrapping (used in Bagging) can help yield better performance since each neural network isn't the same classifier (as compared to some other algorithm such as k-nearest neighbours (k-NN). Hence, averaging our prediction (and rounding it to the closest class) on hidden, unseen data actually helps in generalising the test set performance. For each neural network model, the following hyperparameters are always kept constant:

- Number of hidden layers $= 2$
- Number of hidden neurons in each layer $= 500$
- Number of training iterations/epochs $= 500$
- Adam optimiser (will change hyperparameters for the optimiser but not the optimiser itself)

Apart from using randomised initial weights to induce stochasticity in each neural network model, we also make the following decisions regarding other hyperparameters:

- Adam optimiser – randomly initialise the learning rate $\eta \sim U(0.001, 0.01)$, beta 1 $\beta_1 \sim U(0.85, 0.95)$ and beta 2 $\beta_2 \sim U(0.9, 1)$ for every model where $U(.,.)$ represents the continuous uniform distribution
- For the first three neural networks (models 1-3), use the rectified linear unit (reLU) activation function. For the next three (models 4-6), use the hyperbolic tangent activation function. For the last three (models 7-9), use the sigmoid activation function.

Hence, every base learner is different to the other and is unstable (stochastic) so we expect the generalisation performance of the ensemble to be valid, useful and reliable for classifying on unseen data.

Regarding training/testing splits, first observe that Bagging inherently uses the Bootstrap to generate a resampled version of our modified dataset (first three principal components as input features and meat name as the target). Initially, we first split the entirety of the modified dataset into 67% training ($\frac{2}{3} \times 100\%$) and 33% testing ($\frac{1}{3} \times 100\%$). We keep the testing set completely hidden until each neural network is completely trained (x_{test} and y_{test}). Using the 67% training set, when we perform the Bootstrap, approximately $\frac{1}{e} \times 100\% \approx 37\%$ of this 67% training set don't appear in the resampled dataset which can be used as a validation set (x_{valid} and y_{valid}). The remaining $\left(1 - \frac{1}{e}\right) \times 100\% = \approx 63\%$ of this 67% training set can be relabelled as training data (x_{train} and y_{train}). Why don't we just use the validation set for evaluating the ensemble performance? This is because the validation set is different for every model (depending on the resample) so a common test set is naturally required to compare the ensemble with each of the individual base learners.

Finally, we perform one-hot encoding on each of y_{train}, y_{valid} and y_{test} to be able to train each neural network. For each sample, relabel the target as:

- Beef $\rightarrow 0 \rightarrow [1,0,0,0]$
- Chicken $\rightarrow 1 \rightarrow [0,1,0,0]$
- Lamb $\rightarrow 2 \rightarrow [0,0,1,0]$
- Pork $\rightarrow 3 \rightarrow [0,0,0,1]$

Results:

For the full raw results of each model, please view the output of the "Train and test 9 neural networks" cell in the Jupyter notebook attached at the end of this report.

Descriptive Statistics based on performance on Validation and Test Set				
	Minimum	Maximum	Mean	Standard Deviation
Validation set	62.86%	87.18%	71.81%	7.63%
Testing set	54.87%	78.76%	71.19%	7.58%

Table 1: Descriptive statistics for each neural network model based on validation and test set performance

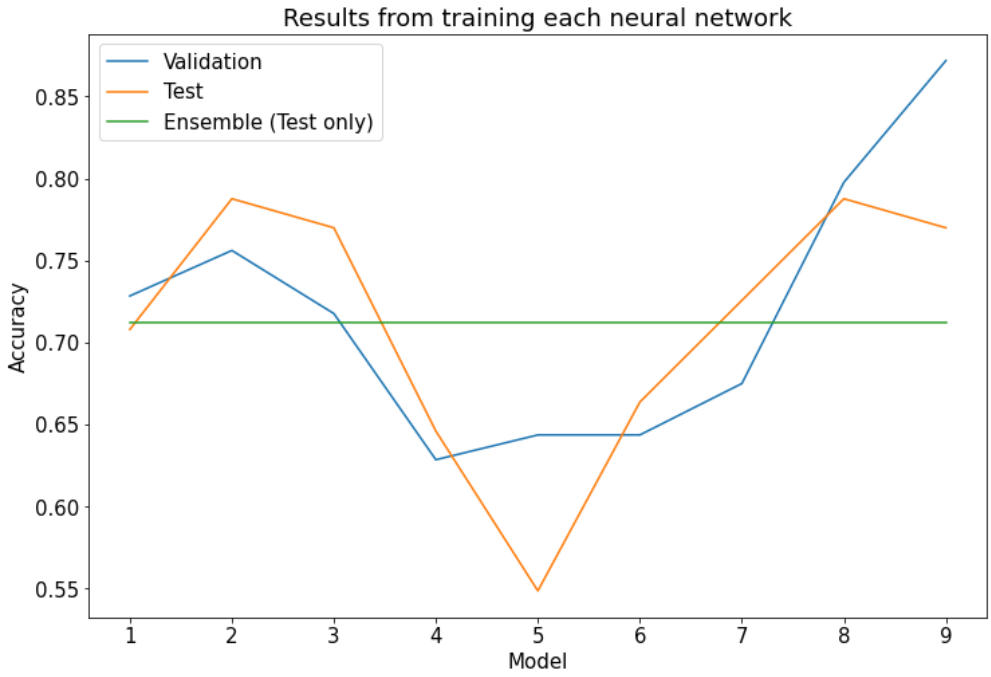


Figure 5: Each model's performance on validation and test sets compared to ensemble performance (on test set ONLY)

The descriptive statistics provided in table 1 for the performance of each neural network on the validation and test sets show minimal difference in mean and standard deviation statistics. However, the minimum and maximums are lower on the test set compared to the validation set even after randomly shuffling the data initially. This may be because the size of the test size (113 samples) is always $\approx \frac{3}{2}$ times greater than the validation set, so it is almost bound to have lower performance on the test set as there is more hidden data to incorrectly misclassify. Overall, these statistics are reasonable because both the validation sets and test sets act as completely hidden, unseen data for each neural network, so both of these form a good estimate of the generalisation performance of each individual base learner.

Recall from above that we needed a separate test set for the purposes of evaluating the ensemble generalisation performance (since the validation sets for each model are different as the bootstrapped datasets are different). From figure 5, we can see how well the set of 10 neural networks generalises on the hidden test data. The predictions made by the ensemble are based on the average prediction rounded up/down to the nearest integer class (beef, chicken, lamb or pork). This yields the same result as using the majority vote but was easier to implement in Python since the raw predictions for each model on the test set via TensorFlow was deemed to be too difficult to obtain.

From figure 5, whilst it appears that a lot of the individual base learners tend to outperform the ensemble, if we were unlucky enough to train a single model and end up with either models 4, 5 or 6 and use that to be a sole indicator of the generalisation performance of that model, it would clearly be undesirable. Whilst it is plausible that the weights in these models may have converged to a local minima due to the combination of random weight initialisation and/or randomised Adam optimiser hyperparameter settings, a more clear explanation is given by the use of the hyperbolic tangent activation function which is common to all these three models (recall our

hyperparameter settings from above). Hence, it is reasonable to assume that the hyperbolic tangent activation function does not perform well when training a neural network for classifying the meats on this reduced dataset. Similarly, in model 9 where the validation performance is excessively higher than the test performance, it appears too good to be true since it outperforms all of the other models. Finally, we can't be certain how any of these individual models would perform on further unseen data since the test set size of 113 samples is quite small, we can only be confident. Our confidence in these estimates increases as N increases (statistical interpretation). Hence, the ensemble outperforms using each of the individual base learners.

Conclusion:

In conclusion, we have analysed the Food Nutrients dataset by first conducting a number of preprocessing steps relevant to the data (random shuffle, removing columns with NaNs, string processing, identifying a target for a machine learning problem, converting categorical to numerical features and performing PCA to reduce dimensionality). Then, we proceeded to train and test 9 neural networks and compare the generalisation performance of these models to an ensemble created by these 9 base learners via bagging. It was deemed that the ensemble generally outperforms each of the base learners. However, it is important to note that ensembles do have downsides – any increases to the general number of hidden layers/neurons or training steps or just training more models can have a huge computational cost and time attached to it. Finally, a potential next step to advance our analysis of the neural network models created using this dataset would be to perform hypothesis testing on the mean of the ensemble performance and generate 95% confidence bounds to obtain a better idea of how well our ensemble generalises to additional unseen data. Confidence bounds are used over confidence intervals since we don't know the true underlying distribution of the ensemble test performance and these can be created using an empirical distribution.

Appendix - Code:

The files $mlp\_model.py$ and $helpers.py$ are similar to the code for Practical 7 (Week 8) but some major changes were made (disabled saving results, TensorBoard and convolutional model, enabled random seed setting, etc.).

1) $mlp\_model.py$:

```
import os
from SupportCode import helpers
from math import sqrt
import random
import numpy as np
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
tf.disable_eager_execution()

def MLPModel(data, dataset, topology={}, optimiser={}, act=tf.nn.relu, max_steps=100, path="",
display_results=False, seed=None):
  # Set seed for reproducable results
  if seed:
    # os.environ['PYTHONHASHSEED'] = str(seed)
    # random.seed(seed)
    # CAUTION - calling function within loop will reset numpy seed everytime and affect RNG for
Q2, Q3, Q4
    # np.random.seed(seed)
    tf.random.set_random_seed(seed)

  # Set up data
  x_train, y_train, x_valid, y_valid, x_test, y_test = data

  # Create inputs
  # tf.reset_default_graph()
  sess = tf.InteractiveSession()
  optimise = helpers.optimiserParams(optimiser)
  if optimise==None:
    print("Invalid Optimiser")
    return
  with tf.name_scope('input'):
    x = tf.placeholder(tf.float32, [None, x_train.shape[1]], name='x-input')
    y_labels = tf.placeholder(tf.float32, [None, y_train.shape[1]], name='y-input')

  # Generate hidden layers
  layers={}
  # Default of 1 hidden layer with 500 neurons if no argument passed to topology parameter
  hiddenDims = topology.setdefault("hiddenDims",[500])
  for i in range(len(hiddenDims)):
    if i==0:
      layers[str(i)] = helpers.FCLayer(x, x_train.shape[1],
hiddenDims[i],"hidden_layer_"+str(i),act=act)
    else:
      layers[str(i)] = helpers.FCLayer(layers[str(i-1)],hiddenDims[i-
1],hiddenDims[i],"hidden_layer_"+str(i),act=act)
  y = helpers.FCLayer(layers[str(i)], hiddenDims[i], y_train.shape[1], 'output_layer',
act=tf.identity)
```

```
with tf.name_scope('cross_entropy'):
  diff = tf.nn.softmax_cross_entropy_with_logits(labels=y_labels, logits=y)
  with tf.name_scope('total'):
    cross_entropy = tf.reduce_mean(diff)
tf.summary.scalar('cross_entropy', cross_entropy)
with tf.name_scope('train'):
  train_step = optimise.minimize(cross_entropy)
with tf.name_scope('accuracy'):
  with tf.name_scope('correct_prediction'):
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_labels, 1))
  with tf.name_scope('accuracy'):
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.summary.scalar('accuracy', accuracy)
merged = tf.summary.merge_all()

# Save results (disabled)
"""
if not path:
  trainPath,testPath = helpers.getSaveDir(dataset, optimiser["optMethod"])
else:
  trainPath,testPath = helpers.getSaveDir(dataset, path)
train_writer = tf.summary.FileWriter(trainPath, sess.graph)
test_writer = tf.summary.FileWriter(testPath)
"""

tf.global_variables_initializer().run()

def feed_dict(train):
  """Make a TensorFlow feed_dict: maps data onto Tensor placeholders."""
  if train:
    # Randomly choose 100 data samples to train on (x_train and y_train)
    idx = np.random.randint(0, len(x_train), 100)
    xs = [x_train[i] for i in idx]
    ys = [y_train[i] for i in idx]

  else:
    xs, ys = x_valid, y_valid
  return {x: xs, y_labels: ys}

for i in range(max_steps):
  try:
    if i % 50 == 0:  # Record summaries and test-set accuracy
      summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
      # Saving test results (disabled)
      # test_writer.add_summary(summary, i)
      print(('Accuracy at step %s: %s' % (i, acc)))
    else:  # Record train set summaries, and train
      if i % 25 == 24:  # Record execution stats
        run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
        run_metadata = tf.RunMetadata()
        summary, _ = sess.run([merged, train_step],
                              feed_dict=feed_dict(True),
                              options=run_options,
                              run_metadata=run_metadata)
        # Save train results (disabled)
        # train_writer.add_run_metadata(run_metadata, 'step%03d' % i)
        # train_writer.add_summary(summary, i)
```

```
        # print(('Adding run metadata for', i))
      else:  # Record a summary
        summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True))
        # Save train results (disabled)
        # train_writer.add_summary(summary, i)
  except:
    print("FAILED DURING TRAINING")
    break

# Close saved results file (disabled)
# train_writer.close()
# test_writer.close()

# Validation set accuracy
valid_accuracy = sess.run(accuracy, feed_dict={x: x_valid, y_labels: y_valid})
print(f"Accuracy on valid set: {valid_accuracy}")
print("\n")

# Also get model test set accuracy and predictions on test set
test_accuracy = sess.run(accuracy, feed_dict={x: x_test, y_labels: y_test})
predictions = sess.run(tf.argmax(y, 1), feed_dict={x: x_test})
sess.close()

# Display results (won't work if results aren't saved)
if display_results:
  # portTrain=8001, portTest=8002
  helpers.openTensorBoard(trainPath,testPath)

return valid_accuracy, test_accuracy, predictions
```

2) *helpers.py*:

```
import os
import webbrowser
import subprocess
import signal
import platform
import re
from math import sqrt
import time
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

def convParams(topology):
  """
  Sets the convNet defaults
  """
  topology.setdefault('convPoolLayers',2)
  topology.setdefault('filterSize',5)
  topology.setdefault('convStride',1)
  topology.setdefault('numFilters',32)
  topology.setdefault('poolK',2)
  topology.setdefault('poolStride',2)
  topology.setdefault('FCLayerSize',1024)
  return topology

def optimiserParams(optDic):
  """
  Sets the default optimisation parameters
  """
  validOptimisers =["GradientDescent","Adam","RMSProp",
  "Momentum","Adagrad"]
  optimisation = {}
  opt=optDic.setdefault('optMethod',"GradientDescent")
  if opt not in validOptimisers:
    return None
  optimisation['learning_rate']=optDic.setdefault('learning_rate',0.001)
  if opt == "GradientDescent":
    optimiser=tf.train.GradientDescentOptimizer(**optimisation)
  elif opt == "Momentum":
    optimisation["momentum"]=optDic.setdefault('momentum',0.9)
    optimiser = tf.train.MomentumOptimizer(**optimisation)
  elif opt=="Adagrad":
    optimisation["initial_accumulator_value"]=optDic.setdefault('initial_accumulator_value',0.1)
    optimiser = tf.train.AdagradOptimizer(**optimisation)
  elif opt=="RMSProp":
    optimisation["momentum"]=optDic.setdefault('momentum',0.0)
    optimisation["decay"]=optDic.setdefault('decay',0.9)
    optimisation["centered"]=optDic.setdefault('centered',False)
    optimiser = tf.train.RMSPropOptimizer(**optimisation)
  elif opt=="Adam":
    optimisation["beta1"]=optDic.setdefault('beta1',0.9)
    optimisation["beta2"]=optDic.setdefault('beta2',0.999)
    optimiser = tf.train.AdamOptimizer(**optimisation)
  return optimiser

def weight_variable(shape):
```

```
  """Create a weight variable with appropriate initialization."""
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  """Create a bias variable with appropriate initialization."""
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

def variable_summaries(var):
  """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
  with tf.name_scope('summaries'):
    mean = tf.reduce_mean(var)
    tf.summary.scalar('mean', mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar('stddev', stddev)
    tf.summary.scalar('max', tf.reduce_max(var))
    tf.summary.scalar('min', tf.reduce_min(var))
    tf.summary.histogram('histogram', var)

def dropoutLayer(inputs):
  with tf.name_scope('Dropout'):
    keep_prob = tf.placeholder(tf.float32)
    tf.summary.scalar('dropout_keep_probability', keep_prob)
    dropped = tf.nn.dropout(inputs, keep_prob)
    return dropped

def FCLayer(input_tensor, input_dim, output_dim, layer_name, act=tf.nn.relu):
  """Reusable code for making a simple neural net layer.
  It does a matrix multiply, bias add, and then uses ReLU to nonlinearize.
  It also sets up name scoping so that the resultant graph is easy to read,
  and adds a number of summary ops.
  """
  # Adding a name scope ensures logical grouping of the layers in the graph.
  with tf.name_scope(layer_name):
    # This Variable will hold the state of the weights for the layer
    with tf.name_scope('weights'):
      weights = weight_variable([input_dim, output_dim])
      variable_summaries(weights)
      # Disabled since conv model disabled
      """
      # Visualize conv1 features
      with tf.variable_scope('heatmap'):
        # scale weights to [0 255] and convert to uint8 (maybe change scaling?)
        x_min = tf.reduce_min(weights)
        x_max = tf.reduce_max(weights)
        weights_0_to_1 = (weights - x_min) / (x_max - x_min)
        weights_0_to_255_uint8 = tf.image.convert_image_dtype (weights_0_to_1, dtype=tf.uint8)
        # to tf.image_summary format [batch_size, height, width, channels]
        weights_transposed = tf.reshape(weights_0_to_255_uint8, [-1,
int(weights_0_to_255_uint8.shape[0]),
          int(weights_0_to_255_uint8.shape[1]), 1])
        # this will display random 3 filters from the 64 in conv1
        tf.summary.image('heatmap', weights_transposed,10)
      """
    with tf.name_scope('biases'):
```

```
      biases = bias_variable([output_dim])
      variable_summaries(biases)
    with tf.name_scope('Wx_plus_b'):
      preactivate = tf.matmul(input_tensor, weights) + biases
      tf.summary.histogram('pre_activations', preactivate)
    activations = act(preactivate, name='activation')
    tf.summary.histogram('activations', activations)
    return activations

def killProcessesOnPorts(portTrain,portTest):
    ports=[str(portTrain),str(portTest)]
    if "Windows" in platform.system():
        popen = subprocess.Popen(['netstat', '-a','-n','-o'],
                             shell=False,
                             stdout=subprocess.PIPE)
    else:
        popen = subprocess.Popen(['netstat', '-lpn'],
                            shell=False,
                            stdout=subprocess.PIPE)
    (data, err) = popen.communicate()
    data = data.decode("utf-8")

    if "Windows" in platform.system():
        for line in data.split('\n'):
            line = line.strip()
            for port in ports:
                if '127.0.0.1:' + port in line and "0.0.0.0:" in line:
                    pid = line.split()[-1]
                    subprocess.Popen(['Taskkill', '/PID', pid, '/F'])
    else:
        pattern = "^tcp.*((?:{0})).* (?P<pid>[0-9]*)/.*$"
        pattern = pattern.format(')|(?:'.join(ports))
        prog = re.compile(pattern)
        for line in data.split('\n'):
            match = re.match(prog, line)
            if match:
                pid = match.group('pid')
                subprocess.Popen(['kill', '-9', pid])

def openTensorBoard(trainPath,testPath,portTrain=8001,portTest=8002):
  urlTrain = 'http://localhost:'+str(portTrain)
  urlTest = 'http://localhost:'+str(portTest)

  killProcessesOnPorts(portTrain,portTest)
  proc = subprocess.Popen(['tensorboard', '--logdir=' + trainPath,'--host=localhost', '--port='
+ str(portTrain)])
  proc2 = subprocess.Popen(['tensorboard', '--logdir=' + testPath,'--host=localhost', '--port='
+ str(portTest)])
  # time.sleep(4)
  webbrowser.open(urlTrain)
  webbrowser.open(urlTest)

def getSaveDir(dataset, path):
  directory = os.path.dirname(os.path.abspath(__file__))
  if "Windows" in platform.system():
    directory="/".join(directory.split("\\"))
  directory=directory.rsplit("/",1)[0] + "/Results/" + dataset + "/" + path + "/"
```

```
  if not os.path.exists(directory):
    os.makedirs(directory)
  dirList = [d for d in os.listdir(directory) if os.path.isdir(os.path.join(directory, d))]
  dirName = "train"
  maxNum = -1
  for d in dirList:
    if dirName in d:
      num = list(map(int, re.findall('\d+', d)))
      if len(num)==1:
          if num[0]>maxNum:
              maxNum=num[0]
  trainDir = directory + dirName + str(maxNum+1) +'/'
  testDir = directory + "test" + str(maxNum+1) + '/'
  return trainDir, testDir

def openTensorBoardAtIndex(dataset, path, ind, portTrain=8001, portTest=8002):
  directory = os.path.dirname(os.path.abspath(__file__))
  if "Windows" in platform.system():
    directory="/".join(directory.split("\\"))
  directory=directory.rsplit("/",1)[0] + "/Results/" + dataset + "/" + path + "/"
  if not os.path.exists(directory):
    print("You need to run at least a model on the given dataset")
    return
  dirList = [d for d in os.listdir(directory) if os.path.isdir(os.path.join(directory, d))]
  dirName = "train"
  foundDir = False
  for d in dirList:
    if dirName in d:
      num = list(map(int, re.findall('\d+', d)))
      if len(num)==1:
          if num[0]==ind:
              foundDir=True
  if not foundDir:
    print("That index does not exist")
    return
  else:
    trainDir = directory + dirName + str(ind) +'/'
    testDir = directory + "test" + str(ind) + '/'
    openTensorBoard(trainDir,testDir,portTrain,portTest)
```

COMP7703 Final Exam Jupyter Notebook

In [1]:
```python
from mlp_model import *
from SupportCode.helpers import *
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Set default figure size when plotting
plt.rcParams["figure.figsize"] = (12,8)
# Set default font size when plotting
plt.rcParams.update({'font.size': 15})
```

```
WARNING:tensorflow:From d:\virtualenv\ml\tensorflow\lib\site-packages\tensorflow\python\compat\v2_compat.py:96:
disable_resource_variables (from tensorflow.python.ops.variable_scope) is deprecated and will be removed in a f
uture version.
Instructions for updating:
non-resource variables are not supported in the long term
```

Deep learning on Food Nutrients Dataset

Stage 1 - Exploratory Data Analysis (EDA)

Setup data

In [2]:
```python
dataset = pd.read_csv("FoodNutrients.csv")
# (N, D) = number of samples x number of features
dataset.shape
```

Out[2]: (1535, 252)

In [3]:
```python
# Preview original dataset
dataset.head(len(dataset))
```

Out[3]:

	Public Food Key	Classification	Food Name	Energy, with dietary fibre	Energy, without dietary fibre	Moisture (water)	Protein	Nitrogen	Total Fat	Ash	...	Proline	Unnamed: 243	Serine	Unnam
0	NaN	NaN	NaN	kJ	kJ	g	g	g	g	g	...	mg/g N	mg	mg/g N	
1	F009117	11101.0	Tea, green, plain, without milk	7	7	95.7	0.2	0.04	0.1	0	...	NaN	NaN	NaN	N
2	F009125	11101.0	Tea, regular, black, brewed from leaf or teaba...	2	2	99.8	0.1	0.02	0	0	...	NaN	NaN	NaN	N
3	F009115	11103.0	Tea, decaffeinated, black, brewed from leaf or...	6	6	99.8	0.1	0.02	0.1	0	...	NaN	NaN	NaN	N
4	F003017	11201.0	Coffee, black, from instant coffee powder	5	4	99.2	0.2	0.03	0	0.1	...	NaN	NaN	NaN	N
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1530	F004196	31502.0	Gelatine, all types	1449	1449	11	84.4	15.2	0.4	1.8	...	821	12479	252	3
1531	F005647	32102.0	Milk, human/breast, mature, fluid	286	286	87.5	1.3	0.2	4.2	0.2	...	NaN	NaN	NaN	N
1532	F003301	34101.0	Crocodile, tail fillet, raw	442	442	76.2	22.5	3.6	1.6	1	...	NaN	NaN	NaN	N
1533	F003299	34101.0	Crocodile, back leg, raw	455	455	76.1	22	3.52	2.2	0.9	...	NaN	NaN	NaN	N
1534	F003300	34101.0	Crocodile, cooked, no added fat	748	748	60.2	37.1	5.93	3.2	1.6	...	NaN	NaN	NaN	N

1535 rows × 252 columns

In [4]:
```python
# Set seed for partial reproducibility of results (everything numpy related)
np.random.seed(42)

# Identifying a unique feature to explore using machine learning techniques later on
target = dataset["Food Name"].values[1:]
# String splitting to only grab first word denoting the major food type (e.g. Milk, Juice, Beef, etc.)
target = [food.split(",")[0] for food in list(target)]

# First drop all columns (axis=1) with NaN values, then drop any remaining samples (axis=1) that have NaN value
dataset.dropna(axis=1, inplace=True)
dataset.dropna(axis=0, inplace=True)

# Remove first index (units for each feature e.g. kJ, g, etc.)
dataset.drop(index=0, inplace=True)

dataset["Food Name"] = target
# Focus only on samples that are classified as a meat - beef, lamb, pork or chicken
dataset = dataset[dataset["Food Name"].isin(["Beef", "Lamb", "Pork", "Chicken"])]

# Numeric elements appear as strings so force them into ints/floats
dataset = dataset.apply(pd.to_numeric, errors="ignore")

# Randomly shuffle entire dataset
dataset = dataset.iloc[np.random.permutation(len(dataset)), :].reset_index(drop=True)

# Preview reduced dataset
dataset.head(len(dataset))
```

Out[4]:

	Energy, with dietary fibre	Energy, without dietary fibre	Moisture (water)	Protein	Nitrogen	Total Fat	Ash	Total dietary fibre	Alcohol	Total sugars	...	Unnamed: 193	Unnamed: 199	Unnamed: 203	Unnamed: 207	Ur
0	738	738	66.9	22.8	3.66	9.5	1.2	0.0	0	0.0	...	13.82	17.27	2.68	0.35	
1	472	470	72.6	19.4	3.10	1.2	3.0	0.2	0	0.2	...	1.18	2.36	1.77	0.21	
2	953	953	58.4	28.7	4.60	12.6	1.1	0.0	0	0.0	...	67.23	91.26	23.20	1.10	
3	573	573	71.1	21.7	3.47	5.5	1.2	0.0	0	0.0	...	18.88	29.65	3.49	0.34	
4	528	528	72.7	21.9	3.50	4.2	1.1	0.0	0	0.0	...	18.97	36.05	9.77	0.48	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
333	602	602	71.6	20.4	3.26	6.9	1.0	0.0	0	0.0	...	32.32	61.42	16.65	0.82	
334	910	910	57.5	29.9	4.79	10.9	1.3	0.0	0	0.0	...	35.34	54.56	7.33	0.62	
335	848	848	64.1	21.2	3.39	13.2	1.1	0.0	0	0.0	...	25.00	27.93	4.37	0.49	
336	1961	1961	38.4	16.3	2.61	45.5	0.9	0.0	0	0.0	...	0.00	43.36	43.36	6.03	
337	480	480	72.8	23.3	3.73	2.3	1.2	0.0	0	0.0	...	24.95	33.13	5.18	0.28	

338 rows × 60 columns

Interpretation of the dataset: the Australian Food Composition Database (previously called NUTTAB) is a reference database that contains data on the nutrient content of Australian foods. It is referred to as a reference database because it contains mostly analysed data. Only a small proportion of data in the database come from other sources such as recipe calculations, food labels, imputing from similar foods or by borrowing from other countries.

Reference

Identify and convert features with categorical data into discrete numerical data (0/1/2/...)

In [5]:
```python
# Assign 0 to beef, 1 to chicken, 2 to lamb and 3 to pork
dataset["Food Name"].unique()
```

Out[5]: array(['Beef', 'Chicken', 'Lamb', 'Pork'], dtype=object)

In [6]:
```python
for column in dataset.columns:
    # If feature has datatype = object --> categorical feature
    if str(dataset[column].dtype) == "object":
        print(f"Converting categorical feature '{column}' into numerical feature...")

        # Number of unique categories in the feature
        unique_categories = dataset[column].unique()

        # Column that stores discretised data
        num_column = []

        # Replace category with discrete number (0/1/2/...) based on number of unique categories present
        for sample in dataset[column]:
            for i in range(dataset[column].nunique()):
                if sample == unique_categories[i]:
                    num_column.append(i)

        dataset[column] = num_column

dataset.head(len(dataset))
```

Converting categorical feature 'Food Name' into numerical feature...

Out[6]:

	Energy, with dietary fibre	Energy, without dietary fibre	Moisture (water)	Protein	Nitrogen	Total Fat	Ash	Total dietary fibre	Alcohol	Total sugars	...	Unnamed: 193	Unnamed: 199	Unnamed: 203	Unnamed: 207	Ur
0	738	738	66.9	22.8	3.66	9.5	1.2	0.0	0	0.0	...	13.82	17.27	2.68	0.35	
1	472	470	72.6	19.4	3.10	1.2	3.0	0.2	0	0.2	...	1.18	2.36	1.77	0.21	
2	953	953	58.4	28.7	4.60	12.6	1.1	0.0	0	0.0	...	67.23	91.26	23.20	1.10	
3	573	573	71.1	21.7	3.47	5.5	1.2	0.0	0	0.0	...	18.88	29.65	3.49	0.34	
4	528	528	72.7	21.9	3.50	4.2	1.1	0.0	0	0.0	...	18.97	36.05	9.77	0.48	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
333	602	602	71.6	20.4	3.26	6.9	1.0	0.0	0	0.0	...	32.32	61.42	16.65	0.82	
334	910	910	57.5	29.9	4.79	10.9	1.3	0.0	0	0.0	...	35.34	54.56	7.33	0.62	
335	848	848	64.1	21.2	3.39	13.2	1.1	0.0	0	0.0	...	25.00	27.93	4.37	0.49	
336	1961	1961	38.4	16.3	2.61	45.5	0.9	0.0	0	0.0	...	0.00	43.36	43.36	6.03	
337	480	480	72.8	23.3	3.73	2.3	1.2	0.0	0	0.0	...	24.95	33.13	5.18	0.28	

338 rows × 60 columns

Scatter matrix of dataset

In [7]:
```python
# Do not uncomment, too many features present at this stage of the code
# fig = sns.pairplot(dataset, hue="Food Name")
# fig.fig.suptitle("Scatter matrix of dataset", y=1.02)
```

In [8]:
```python
# Form inputs and target as numpy arrays
x_data = dataset[dataset.columns[:-1]].values
# Target is "sex" column
y_data = dataset[dataset.columns[-1]].values
```

In [9]:
```python
# Perform one hot encoding to be able to train neural net
temp_y = []
for target in y_data:
    if target == 0:
        temp_y.append([1, 0, 0, 0])
    elif target == 1:
        temp_y.append([0, 1, 0, 0])
    elif target == 2:
        temp_y.append([0, 0, 1, 0])
    else:
        temp_y.append([0, 0, 0, 1])
y_data = np.array(temp_y)
```

Stage 2 - Perform PCA to reduce dimensionality of the dataset

Define function that implements PCA

In [10]:
```python
def pca(x):
    """
    Function that implements Principal Component Analysis (PCA)
    Inputs:
        x = dataset to perform PCA on
    Outputs:
        z = linear projection of x using z = W^T * (x - m)
        W = principal components (eigenvectors of sample covariance matrix) ordered according to lambdas (see b
        lambdas = eigenvalues of sample covariance matrix in descending order
    """

    # Calculating eigenvalues and eigenvectors
    # dim(W) = d x d, if choosing k columns --> d x k
    [lambdas, W] = np.linalg.eig(np.cov(x - np.mean(x), rowvar=False))

    # May be complex due to numerical errors
    lambdas = lambdas.real
    W = W.real

    # Sorting eigenvalues from largest to smallest
    idx = np.argsort(lambdas)[::-1]
    lambdas = np.sort(lambdas)[::-1]

    # Sorting eigenvectors (columns) according to index sequence from sorting eigenvalues
    W = W[:, idx]

    # z = W^T * (x - m)
    z = np.dot(W.T, (x - np.mean(x)).T).T;

    return [z, W, lambdas]
```

In [11]:
```python
# Implement PCA - obtain linear projection of x (z), principal components (W) and eigenvalues (lambdas)
z, W, lambdas = pca(x_data)

# Proportion of variance explained by first two principal components
np.sum(lambdas[0:3])/np.sum(lambdas)
```

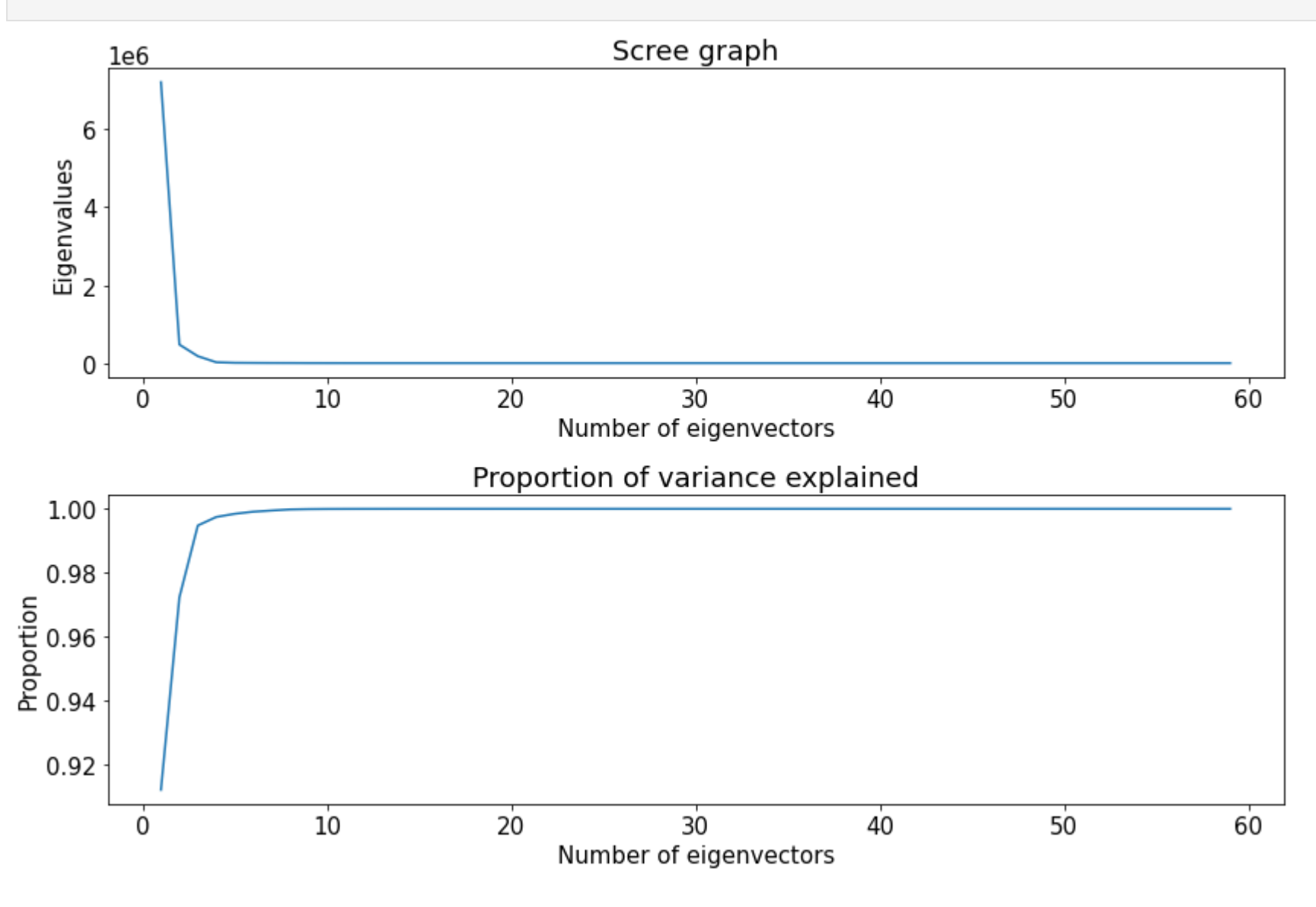
Out[11]: 0.9948050753716449

Scree graph and proportion of variance explained

In [12]:
```python
fig, axs = plt.subplots(2, 1)
axs[0].plot(range(1, len(W) + 1), lambdas)
axs[0].set_xlabel("Number of eigenvectors")
axs[0].set_ylabel("Eigenvalues")
axs[0].set_title("Scree graph")


axs[1].plot(range(1, len(W) + 1), np.cumsum(lambdas)/np.sum(lambdas))
axs[1].set_xlabel("Number of eigenvectors")
axs[1].set_ylabel("Proportion")
axs[1].set_title("Proportion of variance explained")

fig.tight_layout()
```



Scatter matrix of new dataset after PCA

In [13]:
```python
# Keep first k principal components
k = 3

new_dataset = pd.DataFrame(np.concatenate((z[:, range(0, k)], dataset["Food Name"].values.reshape((len(z), 1)))
new_dataset.columns = ["x" + str(i) for i in range(1, k + 1)] + ["Food Name"]
# new_dataset.drop(index=0, inplace=True)
new_dataset.head(len(new_dataset))
```

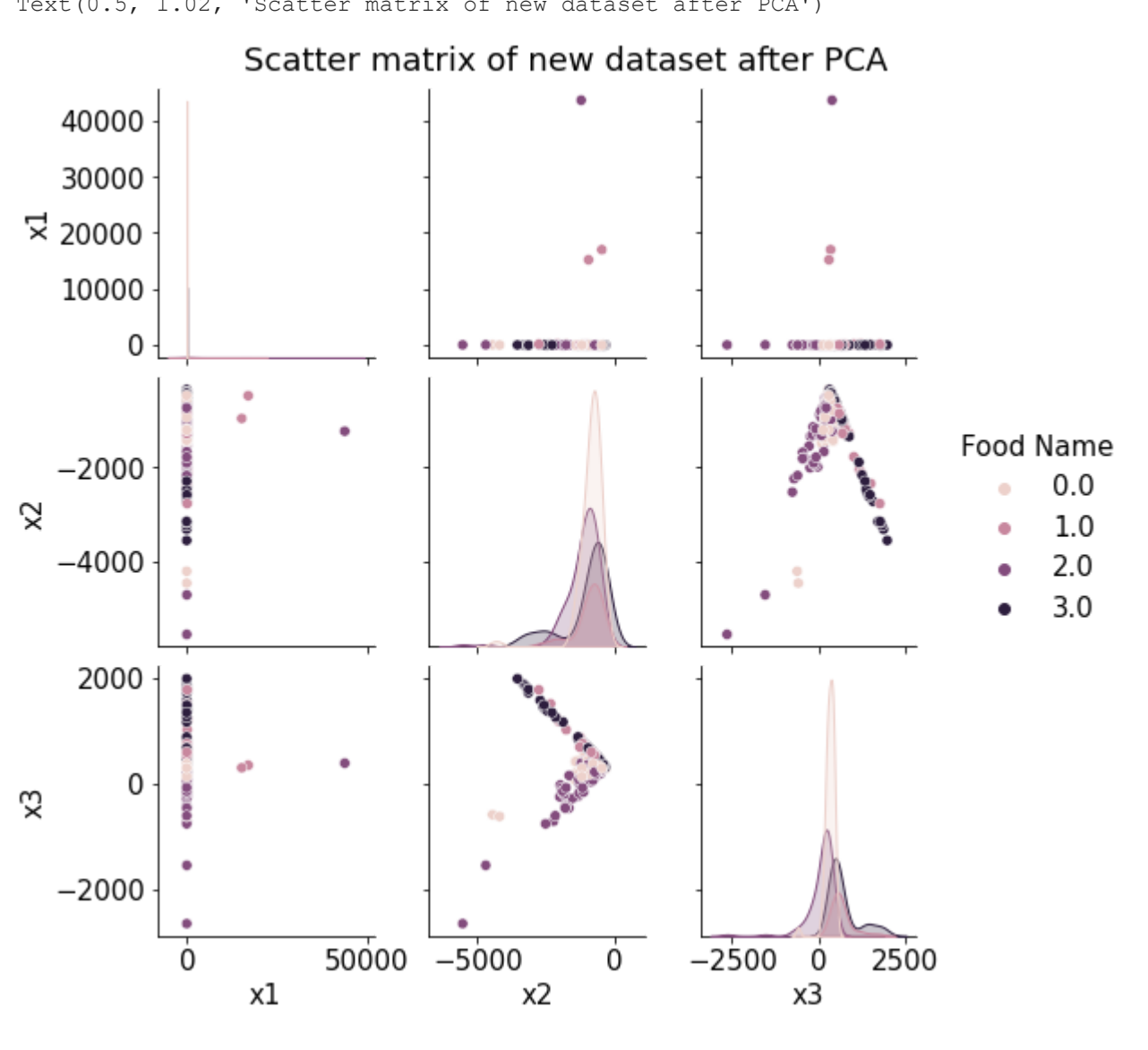
Out[13]:

	x1	x2	x3	Food Name
0	-88.393047	-972.731432	198.292865	0.0
1	-76.944884	-406.144337	363.219672	1.0
2	-82.240439	-1387.184452	115.327520	2.0
3	-91.398701	-665.342333	239.275946	0.0
4	-82.407958	-571.613207	266.018529	2.0
...	...	...	...	...
333	-78.578234	-731.950874	225.141560	2.0
334	-91.506295	-1193.562697	297.573445	0.0
335	-86.234467	-1202.664378	137.121657	0.0
336	-78.251881	-2287.561544	1343.947093	3.0
337	-93.571782	-468.106518	299.853935	0.0

338 rows × 4 columns

In [14]:
```python
fig = sns.pairplot(new_dataset, hue="Food Name")
fig.fig.suptitle("Scatter matrix of new dataset after PCA", y=1.02)
```

Out[14]: Text(0.5, 1.02, 'Scatter matrix of new dataset after PCA')



Stage 3 - Train 10 Neural Networks to get an ensemble of learners

In [15]:
```python
# Form training and testing sets
train_set = new_dataset.iloc[:int(2/3*len(new_dataset))]
test_set = new_dataset.iloc[int(2/3*len(new_dataset)):]

x_test = test_set[test_set.columns[:-1]].values
y_test = test_set[test_set.columns[-1]].values

temp_y = []
# Perform one hot encoding to be able to train neural net
for target in y_test:
    if target == 0:
        temp_y.append([1, 0, 0, 0])
    elif target == 1:
        temp_y.append([0, 1, 0, 0])
    elif target == 2:
        temp_y.append([0, 0, 1, 0])
    else:
        temp_y.append([0, 0, 0, 1])
y_test = np.array(temp_y)
```

Setup network hyperparameters

In [16]:
```python
topology={}
# Use 2 hidden layers with 500 neurons in each layer
topology['hiddenDims'] = [500, 500]

# Training epochs
max_epochs = 500
```

Setup optimiser dictionary

In [17]:
```python
# Optimisation dictionary for Adam optimiser
optDicAdam = {}
optDicAdam["optMethod"] = "Adam"
```

Train and test 9 neural networks

In [18]:
```python
# Store validation and test set accuracies and predictions on test set for each model
valid_results = []
test_results = []
predictions = []

# Number of models to train
n = 9
for i in range(1, n + 1):
    print(f"Model {i}: ")

    optDicAdam["learning_rate"] = np.random.uniform(low=0.001, high=0.01)
    optDicAdam["beta1"] = np.random.uniform(low=0.85, high=0.95)
    optDicAdam["beta2"] = np.random.uniform(low=0.9, high=1.0)

    print(f"HYPERPARAMETERS: learning_rate={optDicAdam['learning_rate']}, beta1={optDicAdam['beta1']}, beta2={o

    if i in range(1, int(n/3) + 1):
        activationFunction = tf.nn.relu
    elif i in range(int(n/3) + 1, int(2/3*n) + 1):
        activationFunction = tf.nn.tanh
    else:
        activationFunction = tf.nn.sigmoid

    # Perform bootstrapping by resampling from new dataset (after PCA)
    rand_idxs = np.random.randint(0, len(train_set), len(train_set))
    bootstrap = train_set.iloc[rand_idxs, :].reset_index(drop=True)

    # Training set = bootstrapped dataset
    x_train = train_set[train_set.columns[:-1]].values
    y_train = train_set[train_set.columns[-1]].values

    temp_y = []
    # Perform one hot encoding to be able to train neural net
    for target in y_train:
        if target == 0:
            temp_y.append([1, 0, 0, 0])
        elif target == 1:
            temp_y.append([0, 1, 0, 0])
        elif target == 2:
            temp_y.append([0, 0, 1, 0])
        else:
            temp_y.append([0, 0, 0, 1])
    y_train = np.array(temp_y)

    # Validation set = everything in new dataset that is not in bootstrapped dataset (approx. 1/e = 37%)
    orig_idxs = []
    for idx in range(0, len(train_set)):
        if idx not in rand_idxs:
            orig_idxs.append(idx)

    x_valid = train_set.iloc[orig_idxs, :-1].values
    y_valid = np.array(train_set.iloc[orig_idxs, -1].values)

    temp_y = []
    # Perform one hot encoding to be able to test neural net
    for target in y_valid:
        if target == 0:
            temp_y.append([1, 0, 0, 0])
        elif target == 1:
            temp_y.append([0, 1, 0, 0])
        elif target == 2:
            temp_y.append([0, 0, 1, 0])
        else:
            temp_y.append([0, 0, 0, 1])
    y_valid = np.array(temp_y)

    data = [x_train, y_train, x_valid, y_valid, x_test, y_test]

    tf.keras.backend.clear_session()

    # Train model using bootstrapped dataset and test on samples from original dataset that did not appear in
    # bootstrapped dataset
    valid_accuracy, test_accuracy, prediction = MLPModel(data, dataset="Food Nutrients", topology=topology,
                                                         optimiser=optDicAdam, act=activationFunction, max_step

    valid_results.append(valid_accuracy)
    test_results.append(test_accuracy)
    predictions.append(prediction)
```

```
Model 1:
HYPERPARAMETERS: learning_rate=0.005975379792171117, beta1=0.8796510143647798, beta2=0.9419780856446277
WARNING:tensorflow:From d:\virtualenv\ml\tensorflow\lib\site-packages\tensorflow\python\util\dispatch.py:201: s
oftmax_cross_entropy_with_logits (from tensorflow.python.ops.nn_ops) is deprecated and will be removed in a fut
ure version.
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See `tf.nn.softmax_cross_entropy_with_logits_v2`.

Accuracy at step 0: 0.18518518
Accuracy at step 50: 0.69135803
Accuracy at step 100: 0.654321
Accuracy at step 150: 0.67901236
Accuracy at step 200: 0.7407407
Accuracy at step 250: 0.69135803
Accuracy at step 300: 0.7160494
Accuracy at step 350: 0.69135803
Accuracy at step 400: 0.7407407
Accuracy at step 450: 0.7777778
Accuracy on valid set: 0.7283950448036194


Model 2:
HYPERPARAMETERS: learning_rate=0.00372271438890719, beta1=0.8740359816488634, beta2=0.9604534626084497
Accuracy at step 0: 0.24390244
Accuracy at step 50: 0.6097561
Accuracy at step 100: 0.5731707
Accuracy at step 150: 0.69512194
Accuracy at step 200: 0.68292683
Accuracy at step 250: 0.63414633
Accuracy at step 300: 0.68292683
Accuracy at step 350: 0.70731705
Accuracy at step 400: 0.7195122
Accuracy at step 450: 0.69512194
Accuracy on valid set: 0.7560975551605225


Model 3:
HYPERPARAMETERS: learning_rate=0.007564197107178119, beta1=0.8916205181266883, beta2=0.9351762772182137
Accuracy at step 0: 0.1764706
Accuracy at step 50: 0.7176471
Accuracy at step 100: 0.7411765
Accuracy at step 150: 0.7176471
Accuracy at step 200: 0.69411767
Accuracy at step 250: 0.7764706
Accuracy at step 300: 0.7764706
Accuracy at step 350: 0.7764706
Accuracy at step 400: 0.7529412
Accuracy at step 450: 0.7176471
Accuracy on valid set: 0.7176470756530762


Model 4:
HYPERPARAMETERS: learning_rate=0.009237393052526435, beta1=0.9333429646764244, beta2=0.9389159044921493
Accuracy at step 0: 0.17142858
Accuracy at step 50: 0.7285714
Accuracy at step 100: 0.62857145
Accuracy at step 150: 0.64285713
Accuracy at step 200: 0.71428573
Accuracy at step 250: 0.71428573
Accuracy at step 300: 0.54285717
Accuracy at step 350: 0.5
Accuracy at step 400: 0.6857143
Accuracy at step 450: 0.5
Accuracy on valid set: 0.6285714507102966


Model 5:
HYPERPARAMETERS: learning_rate=0.007115330359809877, beta1=0.8756065244805192, beta2=0.9749746804430222
Accuracy at step 0: 0.16091955
Accuracy at step 50: 0.6781609
Accuracy at step 100: 0.6551724
Accuracy at step 150: 0.51724136
Accuracy at step 200: 0.6781609
Accuracy at step 250: 0.52873564
Accuracy at step 300: 0.6781609
Accuracy at step 350: 0.62068963
Accuracy at step 400: 0.5862069
Accuracy at step 450: 0.7356322
Accuracy on valid set: 0.6436781883239746


Model 6:
HYPERPARAMETERS: learning_rate=0.007963258353355686, beta1=0.8936056990099216, beta2=0.9256451333753359
Accuracy at step 0: 0.31034482
Accuracy at step 50: 0.7011494
Accuracy at step 100: 0.6781609
Accuracy at step 150: 0.6896552
Accuracy at step 200: 0.6091954
Accuracy at step 250: 0.6551724
Accuracy at step 300: 0.52873564
Accuracy at step 350: 0.6781609
Accuracy at step 400: 0.7241379
Accuracy at step 450: 0.6666667
Accuracy on valid set: 0.6436781883239746


Model 7:
HYPERPARAMETERS: learning_rate=0.008955441512378325, beta1=0.8691998275531987, beta2=0.9958252510535058
Accuracy at step 0: 0.0875
Accuracy at step 50: 0.7125
Accuracy at step 100: 0.6
Accuracy at step 150: 0.7375
Accuracy at step 200: 0.8
Accuracy at step 250: 0.725
Accuracy at step 300: 0.55
Accuracy at step 350: 0.65
Accuracy at step 400: 0.625
Accuracy at step 450: 0.725
Accuracy on valid set: 0.675000011920929


Model 8:
HYPERPARAMETERS: learning_rate=0.0016245878973037732, beta1=0.9354859669737221, beta2=0.9834279578970185
Accuracy at step 0: 0.27380952
Accuracy at step 50: 0.6666667
Accuracy at step 100: 0.72619045
Accuracy at step 150: 0.7380952
Accuracy at step 200: 0.75
Accuracy at step 250: 0.71428573
Accuracy at step 300: 0.71428573
Accuracy at step 350: 0.7619048
Accuracy at step 400: 0.8095238
Accuracy at step 450: 0.75
Accuracy on valid set: 0.7976190447807312


Model 9:
HYPERPARAMETERS: learning_rate=0.0025951414645148663, beta1=0.9477939758837379, beta2=0.92960312909268
Accuracy at step 0: 0.16666667
Accuracy at step 50: 0.7051282
Accuracy at step 100: 0.71794873
Accuracy at step 150: 0.8076923
Accuracy at step 200: 0.71794873
Accuracy at step 250: 0.8717949
Accuracy at step 300: 0.82051283
Accuracy at step 350: 0.84615386
Accuracy at step 400: 0.8333333
Accuracy at step 450: 0.85897434
Accuracy on valid set: 0.8717948794364929
```

Descriptive statistics on validation and test sets

In [19]:
```python
print(f"Validation set results")
print(f"Minimum: {min(valid_results)}")
print(f"Maximum: {max(valid_results)}")
print(f"Mean: {np.mean(valid_results)}")
print(f"Standard deviation: {np.std(valid_results)}")
print("\n")

print(f"Test set results")
print(f"Minimum: {min(test_results)}")
print(f"Maximum: {max(test_results)}")
print(f"Mean: {np.mean(test_results)}")
print(f"Standard deviation: {np.std(test_results)}")
```

```
Validation set results
Minimum: 0.6285714507102966
Maximum: 0.8717948794364929
Mean: 0.7180535197257996
Standard deviation: 0.07633665949106216


Test set results
Minimum: 0.5486725568771362
Maximum: 0.7876105904579163
Mean: 0.7118977308273315
Standard deviation: 0.07579584419727325
```

In [20]:
```python
# Reshape into predictions along rows and models along columns
predictions = np.array(predictions).reshape(len(y_test), n)

# Generate ensemble test accuracy by calculating mean prediction and rounding it up/down (same as majority vote
ensemble_test_accuracy = 0
for i in range(len(predictions)):
    # Calculate mean prediction and round it up/down
    prediction = np.round(np.mean(predictions[i, :]))
    if prediction >= 0.5:
        prediction = 1
    else:
        prediction = 0

    # Ensemble correctly predicted
    if prediction == test_set["Food Name"].iloc[i]:
        ensemble_test_accuracy += 1

ensemble_test_accuracy /= len(predictions)
```

Plot validation and test set results

In [21]:
```python
plt.plot(range(1, n + 1), valid_results, label="Validation")
plt.plot(range(1, n + 1), test_results, label="Test")
plt.plot(range(1, n + 1), [np.mean(test_results)]*n, label="Ensemble (Test only)")
plt.xlabel("Model")
plt.ylabel("Accuracy")
plt.title("Results from training each neural network")
plt.legend()
```

Out[21]: <matplotlib.legend.Legend at 0x2085db90cd0>