

Report For Assignment

Student name: Yuqiao Chen
Email: yuqiao.chen@student.reading.ac.uk
Student Number: 29836354
Module code: CSMAI21
Convenor name: Dr Yevgeniya Kovalchuk
Date: 20/02/2022

Abstract

In this project, we use three different machine learning models for the spam dataset to determine the properties of emails, and one of them is built using Keras. The build of each model is shown step by step in the article, with detailed descriptions of the parameters and ideas. A comparison of the three models will be presented at the end of the article.

1. Background and problem to be addressed

The email have a pivotal role in people's daily communication, but some people also use email for propaganda, advertising, pornography, etc. For individuals, most of these can be regarded as spam emails, i.e., meaningless emails.

We aim to use this spam dataset to effectively identify spam using the machine learning models we have built.

2. Dataset(s) description

The Spam Database is a classic, one-of-a-kind dataset for machine learning classification. The spam collected by the Spam Database comes from the postmasters of the dataset and the individuals who submitted the spam.

The Spambase Dataset is a spam dataset that contains 57 attributes and 4601 instances. The last column indicates whether the email is considered spam (1) or not (0), i.e., unsolicited commercial email. Most attributes suggest whether a particular word or character frequently appears in the email. The run-length attribute (55-57) measures the length of a sequence of consecutive uppercase letters. (Hopkins et al., n.d.)

3. Data visualisation, preprocessing, feature selection

3.1. Get the Data

I choose “Spambase” as my dataset. According to the information of this dataset, we get the column of "word_freq_george" and "word_freq_650" are indicators of non-spam, so in the following steps, I will remove these two columns to avoid affecting the model construction. (*Spambase Data Set*, n.d.)

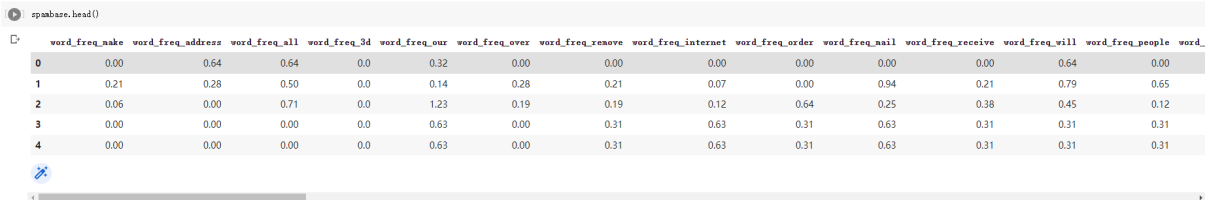
we store the dataset into the value of “spambase”.

```
spambase =  
pd.read_csv('https://datahub.io/machine-learning/spambase/r/spambase.csv'  
)
```

3.2. Preprocessing Data

First of all, we need to check the first five rows of the dataframe. We can see the values of each column(feature values). We need to use these values as input for our model.

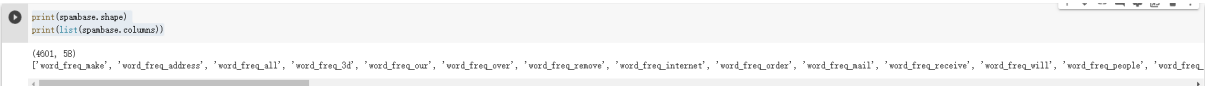
```
spambase.head()
```



	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_out	word_freq_over	word_freq_remove	word_freq_internet	word_freq_order	word_freq_mail	word_freq_receive	word_freq_will	word_freq_people	word_freq
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00	0.00	0.00	0.00	0.64	0.00	
1	0.21	0.28	0.50	0.0	0.14	0.28	0.21	0.07	0.00	0.94	0.21	0.79	0.65	
2	0.06	0.00	0.71	0.0	1.23	0.19	0.19	0.12	0.64	0.25	0.38	0.45	0.12	
3	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	0.31	0.63	0.31	0.31	0.31	
4	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	0.31	0.63	0.31	0.31	0.31	

Then, check the shape of the dataframe. We have 58 columns, and one of them is “class”, the others are features. Each feature has 4601 items.

```
print(spambase.shape)
print(list(spambase.columns))
```

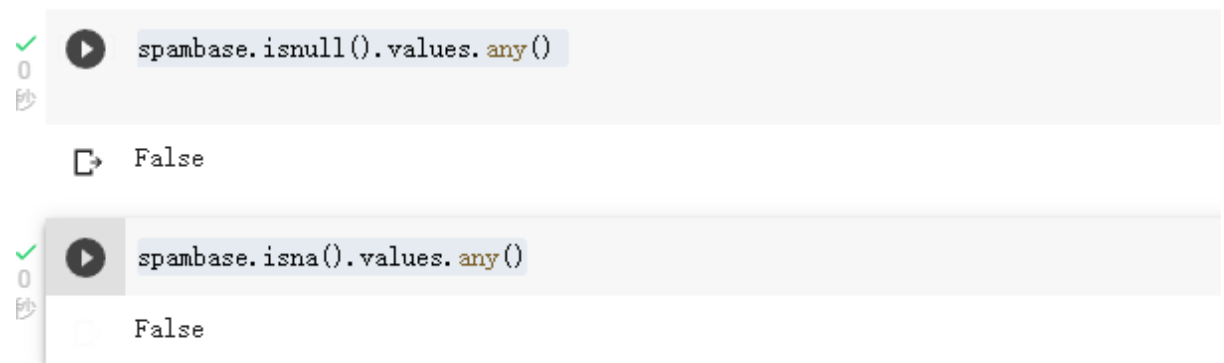


```
(4601, 58)
['word_freq_make', 'word_freq_address', 'word_freq_all', 'word_freq_3d', 'word_freq_out', 'word_freq_over', 'word_freq_remove', 'word_freq_internet', 'word_freq_order', 'word_freq_mail', 'word_freq_receive', 'word_freq_will', 'word_freq_people', 'word_freq']
```

Then, check the dataframe to see whether it has the Nan and missing value. In the end, there are no Nan values and missing values, which is a great dataset.

```
spambase.isnull().values.any()
spambase.isna().values.any()
```

Check the whole dataset whether has Null and missing values.



```
spambase.isnull().values.any()
```

```
False
```

```
spambase.isna().values.any()
```

```
False
```

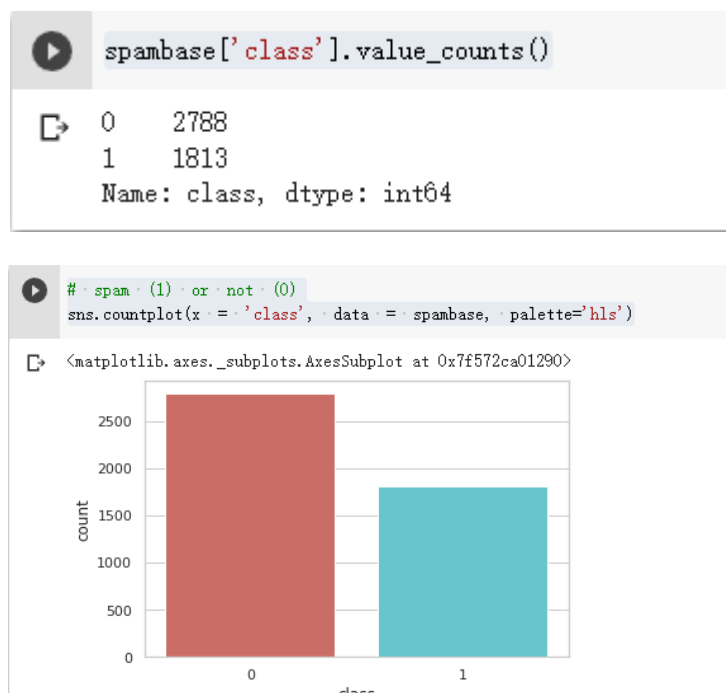
3.3. Data exploration

After we had checked that there were no significant problems with the dataframe, we came to the stage of exploring the dataset.

Firstly, we all know if the email should be spam or not, so we have two classes and use "value_counts" to check the number of each type.

```
spambase['class'].value_counts()

# spam (1) or not (0)
sns.countplot(x = 'class', data = spambase, palette='hls')
```



We have 2788 regular emails and 1813 spam. We can see the number of 2 types are not equal. Spam is almost two-thirds of regular mail.

Then we calculate each type to make statistics.

```
# spam (1) or not (0)
# count the number of spam and not spam
count_spam = len(spambase[spambase['class']==1])
count_no_spam = len(spambase[spambase['class']==0])

# calculate the percentage of each type and print them.
pct_of_spam = count_spam/(count_no_spam+count_spam)
print("percentage of type spam", pct_of_spam*100)
```

```
pct_of_no_spam = count_no_spam/(count_no_spam+count_spam)
print("percentage of type no spam", pct_of_no_spam*100)
```

Then we get the result:

```
percentage of type spam 39.404477287546186
percentage of type no spam 60.59552271245382
```

The result is quite the same as what we guessed before. Spam is almost two-thirds of regular mail.

Then calculate the mean values of each feature by two classes, and we can see some mean values of features are quite smaller than the other features.

```
spambase.groupby("class").mean()
```

We get:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_out	word_freq_over	word_freq_remove	word_freq_internet	word_freq_order	word_freq_mail	word_freq_receive	word_freq_will	word_freq_people
class													
0	0.073479	0.244466	0.200581	0.000886	0.181040	0.044544	0.009383	0.038415	0.038049	0.167170	0.021711	0.536324	0.061664
1	0.152339	0.164650	0.403795	0.164672	0.513955	0.174876	0.275405	0.208141	0.170061	0.350507	0.118434	0.549972	0.143547

3.4. Feature Selection

First, we remove the two previously mentioned columns. Then we divide the entire dataframe into "X" and "y", which we will use later in the plotting and model training. The "X" represents the input features, and the "y" represents our target column.

```
spambase.drop(columns=["word_freq_george", "word_freq_650"], axis =1,
inplace=True)

# Based on the features, we select the input elements and targets.
X = spambase.iloc[:,0:55]
print(X)

y = spambase.iloc[:,55]
print(y)
```

3.5. Data visualisation

Use the “describe()” function to see the whole dataframe:

```
spambase.describe()
```

We get the information like the figure below:

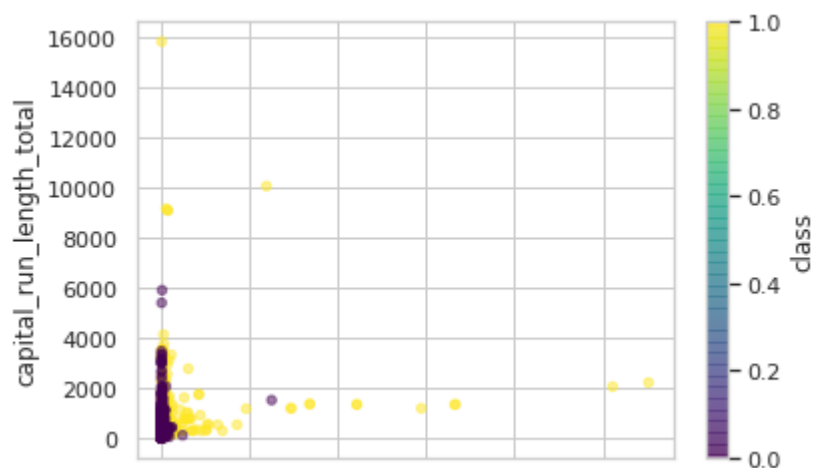
	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	word_freq_order	word_freq_mail	word_freq_receive	word_freq_will	word_freq_people
count	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000
mean	0.104553	0.213015	0.280656	0.065425	0.312223	0.095901	0.114208	0.105295	0.090067	0.239413	0.059824	0.541702	0.093930
std	0.305358	1.290575	0.504143	1.395151	0.672513	0.273824	0.391441	0.401071	0.278616	0.644755	0.201545	0.861698	0.301036
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.100000	0.000000
75%	0.000000	0.000000	0.420000	0.000000	0.380000	0.000000	0.000000	0.000000	0.000000	0.160000	0.000000	0.800000	0.000000
max	4.540000	14.280000	5.100000	42.810000	10.000000	5.880000	7.270000	11.110000	5.260000	18.180000	2.610000	9.670000	5.550000

According to this figure, We can tell that some of the eigenvalues are obvious, and others have such small values and variances that they have little effect on the results.

3.5.1. Scatter Plot

Based on the observation of the overall dataframe description in the previous step, we chose "capital_run_length_average" and "capital_run_length_total" because, on the one hand, most spam emails use many capital letters for emphasis and eye-catching purposes, and, on the other hand, these two columns have a more pronounced mean and variance than the other features.

```
spambase.plot.scatter(x='capital_run_length_average',
y='capital_run_length_total', c='class', alpha = 0.5, cmap='viridis')
```



3.6. Decision Tree

The higher values of the characteristics "capital_run_length_average" and "capital_run_length_total" seem to be strong spam indicators. Furthermore, looking at the scatter plot of these two variables, it is clear that most amateur emails are concentrated at the zero point in the plot above.

Next, we tried to see how all 55 variables affected the final results. We did not have to manually do this tedious task using the well-known CART algorithm to generate a decision tree and rank the features according to their relative importance.

The CART algorithm is a classification algorithm required to build a decision tree based on Gini's impurity index. It is an essential machine learning algorithm and provides various use cases. A statistician named Leo Breiman coined the phrase to describe Decision Tree algorithms that may be used for classification or regression predictive modelling issues. " (*A Classification and Regression Tree (CART) Algorithm*, 2021)

An example of a partially grown SpamBase decision tree is as follows:

First, we create a decision tree classifier by using the sklearn library. We set the max_depth of the tree for 3 Because we want to identify which features are more critical and do not want to make the tree too deep and thus make it very cumbersome for us to identify.

```
clf = DecisionTreeClassifier(random_state=0, splitter="best",
max_depth=3).fit(X,y)
```

Here we use an open-source tool, "dtreeviz", to virtualise our decision tree classifier, which is a powerful tool for seeing the tree. (*Parrrt/dtreeviz: A Python Library for Decision Tree Visualization and Model Interpretation.*, n.d.)

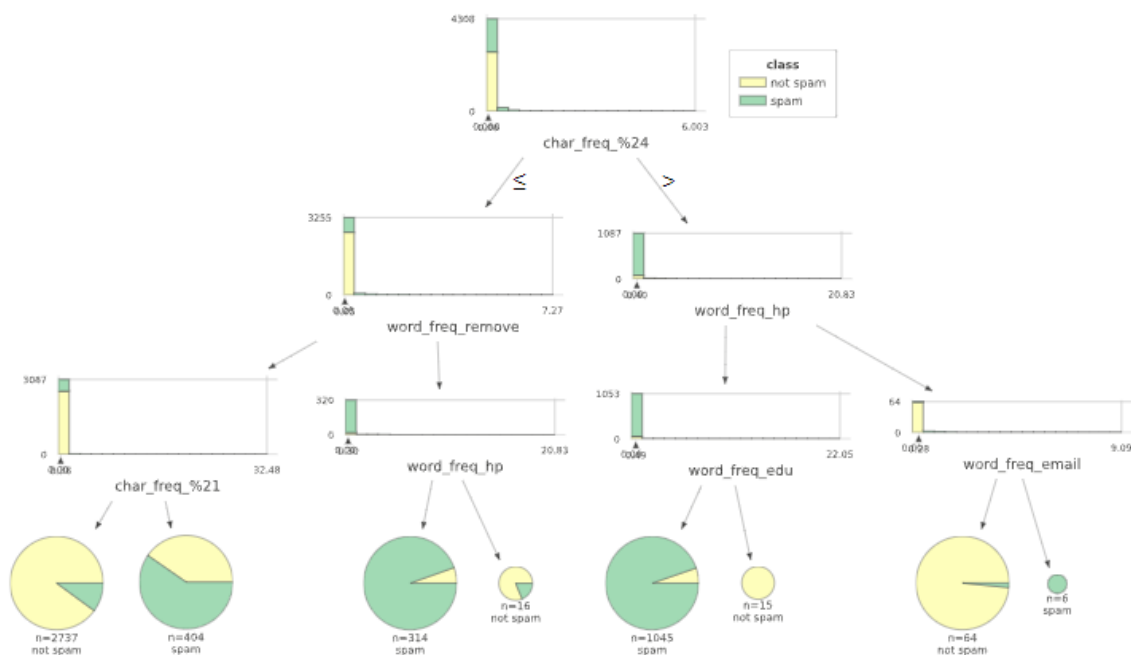
```
viz = dtreeviz(clf, X, y,
               target_name="class",
               feature_names=list(spambase),
               class_names=['not spam', 'spam']
               )

viz
```

As we can see, the tree is like the figure below.

We can see that the decision tree looks at features such as "%24", "hp", and "remove". This way, we can see which features are most important and which parts are redundant for spam filtering.

By averaging more trees, we can calculate the relative importance of features by counting the number of times they appear in each tree.



Using this method, we can generate the following histogram.

```
from sklearn.inspection import permutation_importance

# get the feature name of dataframe(Columns Names)
feature_names = [f"{i}" for i in X]

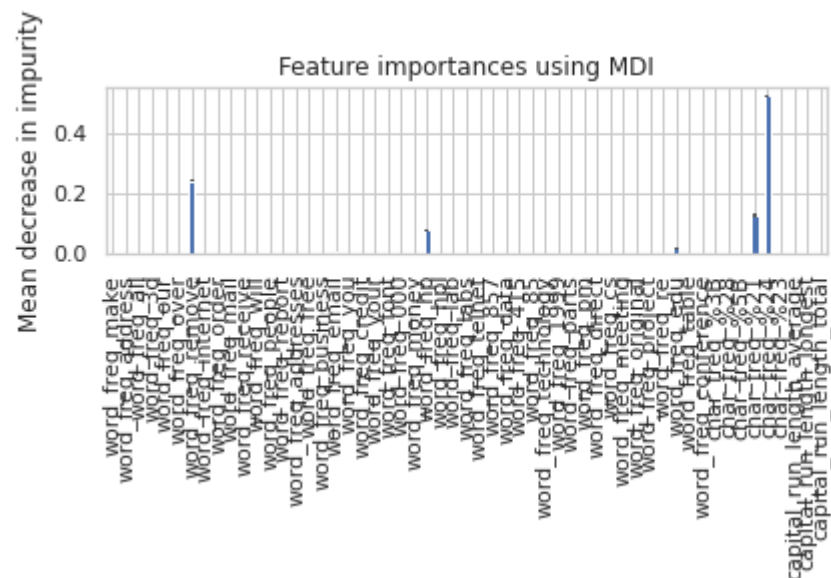
# Associating feature names with the importance of the feature.
forest_importances = pd.Series(clf.feature_importances_,
index=feature_names)

# split the data into train set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

# Permutation importance for feature evaluation
result = permutation_importance(
    clf, X_train, y_train, n_repeats=10, random_state=42, n_jobs=2
)

fig, ax = plt.subplots()
forest_importances.plot.bar(yerr=result.importances_std, ax=ax)
ax.set_title("Feature importances using MDI")
ax.set_ylabel("Mean decrease in purity")
fig.tight_layout()
```


In this section, we use "Feature importance based on feature permutation" permutation feature importance overcomes the limitations of impurity-based feature importance: they are not biased towards high base features and can be computed on missed test sets. Full permutation importance is more costly to compute. A feature is shuffled n times, and the model is refitted to estimate its importance. For more details, see Ranking feature importance. We can now plot importance rankings.



3.7. Visualise the more essential dimensions

Scatter matrix is a powerful way to judge the importance between two features, “In multivariate statistics and probability theory, the scatter matrix is a statistic used to make estimates of the covariance matrix”. (*Scatter Matrix*, n.d.)

The interrelationship between several variables can be seen in the output image, which variables influence the determination of spam.

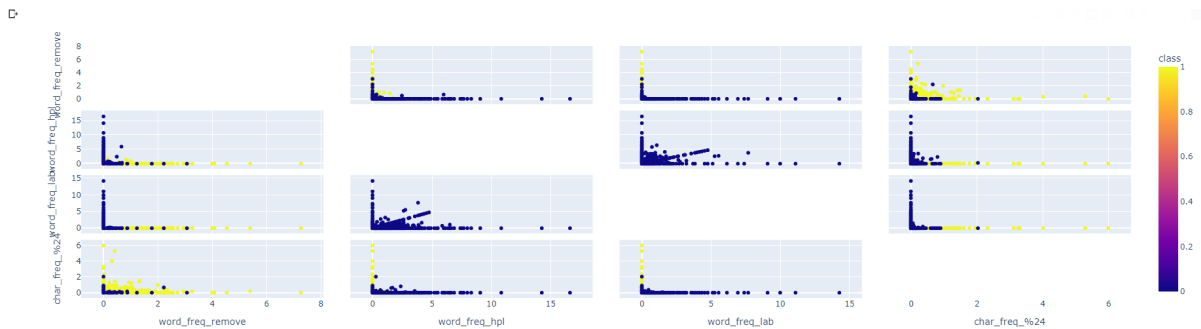
```
fig = px.scatter_matrix(
    spambase,
    dimensions=["word_freq_remove", "word_freq_hp1", "word_freq_lab",
"char_freq %24"],
```

```

    color="class"
)
fig.update_traces(diagonal_visible=False)
fig.show()

```

From the results, the higher the value of "world_freq_remove" and "char_freq_%24", the higher the number of spam messages.



3.8. PCA

PCA is also an excellent tool to explore our dataset.

“Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest. PCA is used in exploratory data analysis and for making predictive models. It is commonly used for dimensionality reduction.” (*Principal Component Analysis*, n.d.)

3.8.1. 2D PCA Scatter Plot

First, we use a 2D scatter plot to plot the data after PCA.

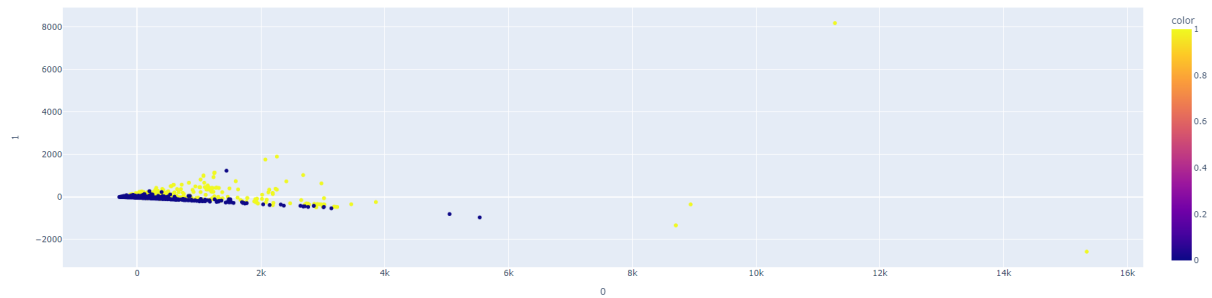
```

# set the components to 2 because we need a 2D plot.
pca_2D = decomposition.PCA(n_components=2)
X_pca = pca_2D.fit_transform(X)

fig = px.scatter(X_pca, x=0, y=1, color=spambase['class'])
fig.show()

```

Then we get the output figure. We can see that there is still a clear dividing line between spam and non-spam under the two-dimensional image after PCA.



3.8.2. 3D PCA Scatter Plot

To present the data in a more three-dimensional way, we decided to use a three-dimensional scatter plot to present the data, and since we used three dimensions, the components in PCA were set to 3.

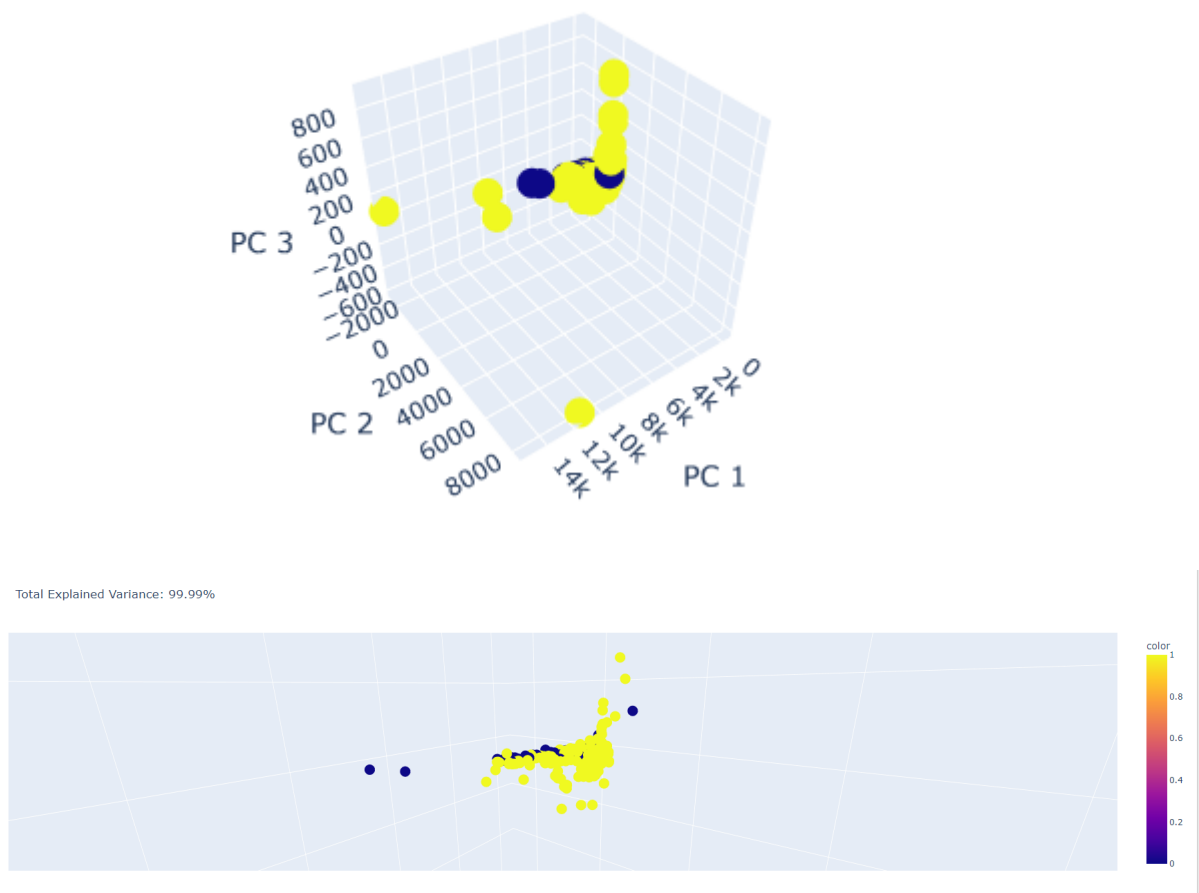
```
# Build the PCA model
pca_3D = decomposition.PCA(n_components=3)

# make the PCA model fit the feature set
components = pca_3D.fit_transform(X)

# Percentage of variance explained by each of the selected components.
total_var = pca_3D.explained_variance_ratio_.sum() * 100

fig = px.scatter_3d(
    components, x=0, y=1, z=2, color=spambase['class'],
    title=f'Total Explained Variance: {total_var:.2f}%',
    labels={'0': 'PC 1', '1': 'PC 2', '2': 'PC 3'}
)
fig.show()
```

Then we get a 3D to scatter plot. The dividing line for determining spam can be seen clearly in three dimensions by zooming in and rotating.



3.9. MDS

“Multidimensional scaling (MDS) is a means of visualising the level of similarity of individual cases of a dataset.” (*Multidimensional Scaling*, n.d.)

Multidimensional scaling is an exploratory process by which we can reduce the number of features before proceeding with MDS. We should normalise the data. Observed; if possible, reveal existing structure in the data; reveal relevant features.

To speed things up, we used "mdscuda" to speed things up, which is about 45 times faster than the regular sklearn library. (SethEBaldwin, n.d.)

Before proceeding with MDS, we should normalise the data.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
X_scaled = scaler.fit_transform(X)
```

```
from mdscuda import MDS, mds_fit, minkowski_pairs

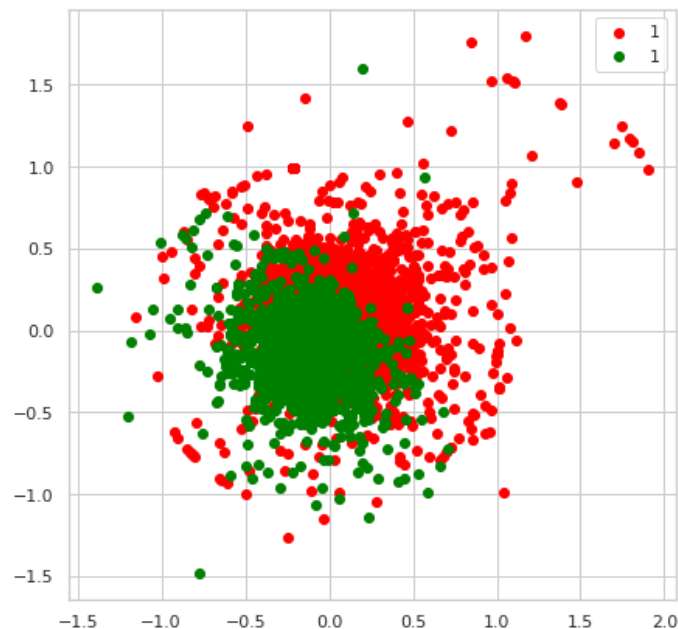
# this returns a matrix of pairwise distances in longform
DELTA = minkowski_pairs(X_scaled, sqform = False)

x = mds_fit(DELTA, n_dims = 2, verbosity = 1)

colors = ['red', 'green']
plt.rcParams['figure.figsize'] = [7, 7]
plt.rc('font', size=14)

for i in np.unique(y):
    subset = x[y == i]

    a = [row[0] for row in subset]
    b = [row[1] for row in subset]
    plt.scatter(a,b,c=colors[i],label=y[i])
plt.legend()
plt.show()
```



Then we get the MDS data. The additional functionality does not significantly improve the performance of the MDS. The MDS provides optimal separation of the two classes.

4. Machine learning model 1 (Logistic Regression)

4.1. Summary of the approach

When selecting the first model, we choose the Logistic Regression as our first model. "In statistics, the logistic model (or logit model) is used to model the probability of a certain class or event existing such as pass/fail, win/lose, alive/dead or healthy/sick. " ("Logistic regression"). According to the question, we want to address whether an email is a spam.

Since determining whether an email is spam is a binary classification problem, we can use logistic regression as our model. "In a binary logistic regression model, the dependent variable has two levels (categorical)." ("Logistic regression")

First, we will normalise the data, then use the sklearn library to initialise the logistic regression model and fit the dataset. After obtaining the model's accuracy, 10-fold cross-validation is used to calculate a more average accuracy of the model and a confusion matrix is used to view the model. Finally, various hyperparameters were adjusted to optimise the model.

4.2. Model training and evaluation

First, we split the data and built a logistic regression model by sklearn library.

```
# split the data into train set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

# iter should bigger than 1000, otherwise the lbfgs failed to converge
LR = LogisticRegression(multi_class='ovr', solver='lbfgs',
max_iter=10000).fit(X_train, y_train)
```

The "score" function is then used to test how accurate the model is.

```
# Train the model using the training sets
LR_score = LR.score(X_test, y_test)
print('The model score is: ', LR_score)
```

Next we get the score of the model. The score function returns the mean accuracy on the given test data and labels. We get 0.90 for our model, which is quite great. Because the score is between 0 and 1, the closer it is to 1, the higher the accuracy and the better the result.

```
The model score is: 0.9080376538740044
```

Next, we print out the coefficients of the model, the mean squared error and the coefficient of determination.

```
# Make predictions using the testing set
y_pred = LR.predict(X_test)
print(y_pred, y_pred.shape)

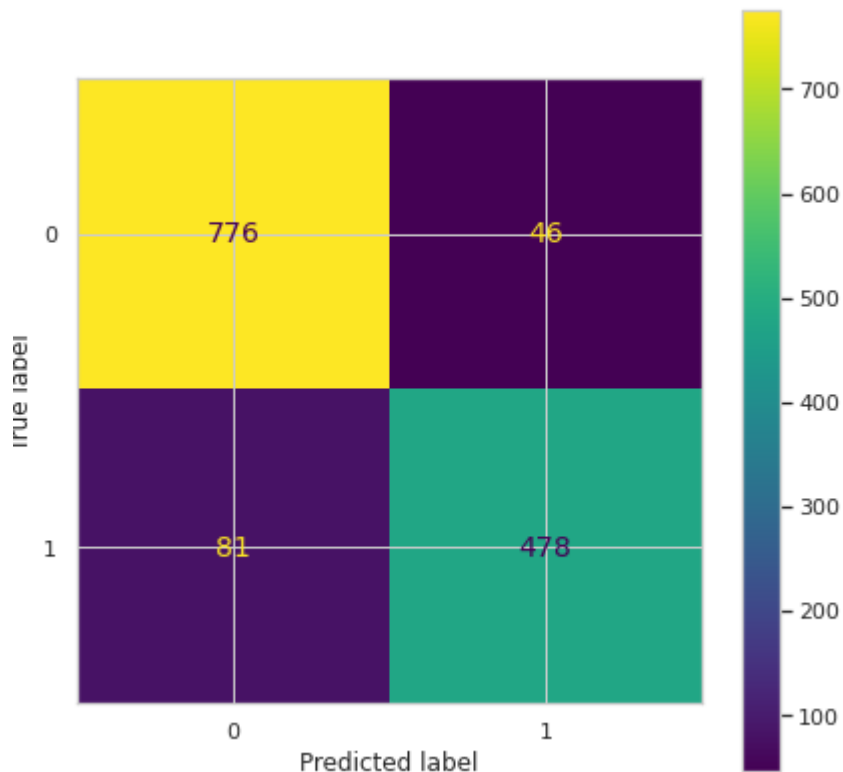
# The coefficients
print("Coefficients: \n", LR.coef_)
# The mean squared error
print("Mean squared error: %.2f" % mean_squared_error(y_test, y_pred))
# The coefficient of determination: 1 is perfect prediction
print("Coefficient of determination: %.2f" % r2_score(y_test, y_pred))
```

We can get this information. Next, we print out the coefficients of the model, the mean squared error and the coefficient of determination. The mean squared error is small and the coefficient of determination is moderately large, greater than 0.25, indicating that the overall model is reliable.

```
Coefficients:
[[-1.51294680e-01 -1.31911131e-01  1.76123864e-01  9.59920025e-01
  7.02273596e-01  3.97828104e-01  2.68092240e+00  6.00390122e-01
  4.98744512e-01  5.47803189e-02  2.54824636e-01 -7.58692936e-02
 -7.13793043e-02  7.20172713e-02  5.29898253e-01  4.01692336e-01
  7.77741989e-01  1.04706193e-01  1.27639781e-01  8.39326645e-01
  2.90144569e-01  2.87969405e-01  2.38859344e+00  4.68074289e-01
 -1.66962172e+00 -1.24742680e+00 -1.28349503e+00 -4.69884504e-01
 -1.29713950e+00 -6.22186388e-01 -1.03476815e+00 -1.12143524e+00
 -1.49901646e+00  1.31671371e+00 -1.63638648e-02  1.44315591e-01
 -5.47612631e-01 -4.82135888e-01 -1.15309241e+00 -1.76484171e+00
 -1.02219993e+00 -1.08340297e+00 -7.32726301e-01 -1.58043438e+00
 -9.59176495e-01 -1.79284531e+00 -1.09144465e+00  1.21548860e-01
 -6.19635086e-01  5.17232017e-01  4.03587023e+00  1.23564972e+00
 -1.43806867e-02  7.75810630e-03  8.80011537e-04]]
Mean squared error: 0.09
Coefficient of determination: 0.62
```

Next, we use the confusion matrix to observe the entire model.

```
cm = confusion_matrix(y_test, y_pred, labels=LR.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=LR.classes_)
disp.plot()
plt.show()
```



4.3. Hyperparameter Tuning

4.3.1. k-fold crossValidation

To prevent overfitting, I use 10-fold cross-validation. Since I am using a binary classification problem in my experiments, "f1" is used as the scoring for cross-validation.

The mean scores and 95% confidence interval of scores are about 0.89.

```
from sklearn.model_selection import cross_val_score
LR_scores = cross_val_score(LR, X, y, cv=10, scoring='f1')
LR_scores

# Mean of scores and 95% confidence interval of scores:
print("Accuracy: %0.2f (+/- %0.2f)" % (LR_scores.mean(), LR_scores.std() * 
```


2))

Accuracy: 0.89 (+/- 0.07)

4.3.2. Hyperparameter adjustment

4.3.2.1. Test different solvers

in this subsection; we change the solver to test whether it will change. The options are "newton-cg", "liblinear", "sag", "saga ". In the results, we can see that "liblinear" has a higher accuracy rate.

```
# lbfgs as solver.
LR_newton_cg = LogisticRegression(multi_class='ovr', solver='newton-cg', max_iter=10000).fit(X_train, y_train)
print('score for solver= newton-cg is: ', LR_newton_cg.score(X_test,y_test))
```

```
score for solver= newton-cg is: 0.9080376538740044
```

```
[ ] # liblinear as solver.
LR_liblinear = LogisticRegression(multi_class='ovr', solver='liblinear', max_iter=10000).fit(X_train, y_train)
print('score for solver= liblinear is: ', LR_liblinear.score(X_test,y_test))
```

```
score for solver= liblinear is: 0.9102099927588704
```

```
[ ] # sag as solver.
LR_sag = LogisticRegression(multi_class='ovr', solver='sag', max_iter=10000).fit(X_train, y_train)
print('score for solver= sag is: ', LR_sag.score(X_test,y_test))
```

```
score for solver= sag is: 0.8189717595944968
```

```
[ ] # saga as solver.
LR_saga = LogisticRegression(multi_class='ovr', solver='saga', max_iter=10000).fit(X_train, y_train)
print('score for solver= saga is: ', LR_saga.score(X_test,y_test))
```

```
score for solver= saga is: 0.8081100651701666
```

4.3.2.2. Test different loss functions

To test whether different loss functions impact the overall model, we chose "saga" as our solver, as saga can use four types of loss functions each.

As we can see in the results, the loss function has little effect on the model's overall accuracy, which remains almost unchanged.

```
[ ] LR_l1 = LogisticRegression(multi_class='ovr', penalty='l1', solver='saga', max_iter=10000).fit(X_train, y_train)
print('score for penalty= l1 is: ', LR_l1.score(X_test,y_test))

score for penalty= l1 is: 0.8081100651701666

[ ] LR_l2 = LogisticRegression(multi_class='ovr', penalty='l2', solver='saga', max_iter=10000).fit(X_train, y_train)
print('score for penalty= l2 is: ', LR_l2.score(X_test,y_test))

score for penalty= l2 is: 0.8081100651701666

[ ] LR_none = LogisticRegression(multi_class='ovr', penalty='none', solver='saga', max_iter=10000).fit(X_train, y_train)
print('score for penalty= l2 is: ', LR_none.score(X_test,y_test))

score for penalty= l2 is: 0.8081100651701666

[ ] # combine l1 and l2 together with 50% each.
LR_elasticnet = LogisticRegression(multi_class='ovr', penalty='elasticnet', solver='saga', l1_ratio=0.5, max_iter=10000).fit(X_train, y_train)
print('score for penalty= l2 is: ', LR_elasticnet.score(X_test,y_test))

score for penalty= l2 is: 0.8081100651701666
```

4.4. Results and discussion (supported with tables/figures & Python snippets)

In the overall comparison, we can see that by using the logistic regression classifier for dichotomous classification, it still has a good correct rate, and with proper adjustment of the solver and loss function, the score can reach as high as 0.9, and we look forward to comparing it with the latter two functions.

5. Machine learning model 2 (Support Vector Machines)

5.1. Summary of the approach

In machine learning, support vector machines is supervised learning models for data analysis in classification and regression with associated learning algorithms. Given a set of training instances, each of which is labelled as belonging to one or the other of two categories, the SVM training algorithm builds a model that assigns new instances to one of the two categories, making it a non-probabilistic binary linear classifier. The SVM model represents instances as points in space, such that the mapping makes the separate categories of instances separated by distinct intervals that are as wide as possible. The new instances are then mapped into the same space, and their class is predicted based on which side of the interval they fall.

First, we will normalise the data, then use the sklearn library to initialise the SVM model and fit the dataset. After getting the model's accuracy, 10-fold cross-validation is used to calculate a more average accuracy of the model. Finally, a confusion matrix is used to view the model.

5.2. Model training and evaluation

We need to normalise the data features for support vector machines before constructing the model.

Not normalising the features does not significantly impact the final experimental results. However, if the difference in the magnitude of the features is too large, we are likely to get poor test results.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

scaler = StandardScaler().fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train), index=y_train)
X_test = pd.DataFrame(scaler.transform(X_test), index=y_test)
```

Next, we will use the normalised data to complete the model.

```
svc = SVC(kernel='linear', gamma='auto').fit(X_train, y_train)
print('SVM Accuracy: {acc:.4f}'.format(acc=svc.score(X_test, y_test)))
```

After building the model, the first thing we need to do is to check the score of the whole model to see the overall performance of the model. We can see the model has a 0.90 score, which is quite around the logistic regression classifier.

```
SVM Accuracy: 0.9088
```

5.2.1. 10-fold cross-validation

In order to prevent the overfitting, we still use the cross validation to test the model.

```
X_std = pd.DataFrame(scaler.transform(X), index=y)

from sklearn.model_selection import cross_val_score
svc_scores = cross_val_score(svc, X_std, y, cv=10, scoring='f1')
svc_scores
```

```
# Mean of scores and 95% confidence interval of scores:  
print("Accuracy: %0.2f (+/- %0.2f)" % (svc_scores.mean(),  
svc_scores.std() * 2))
```

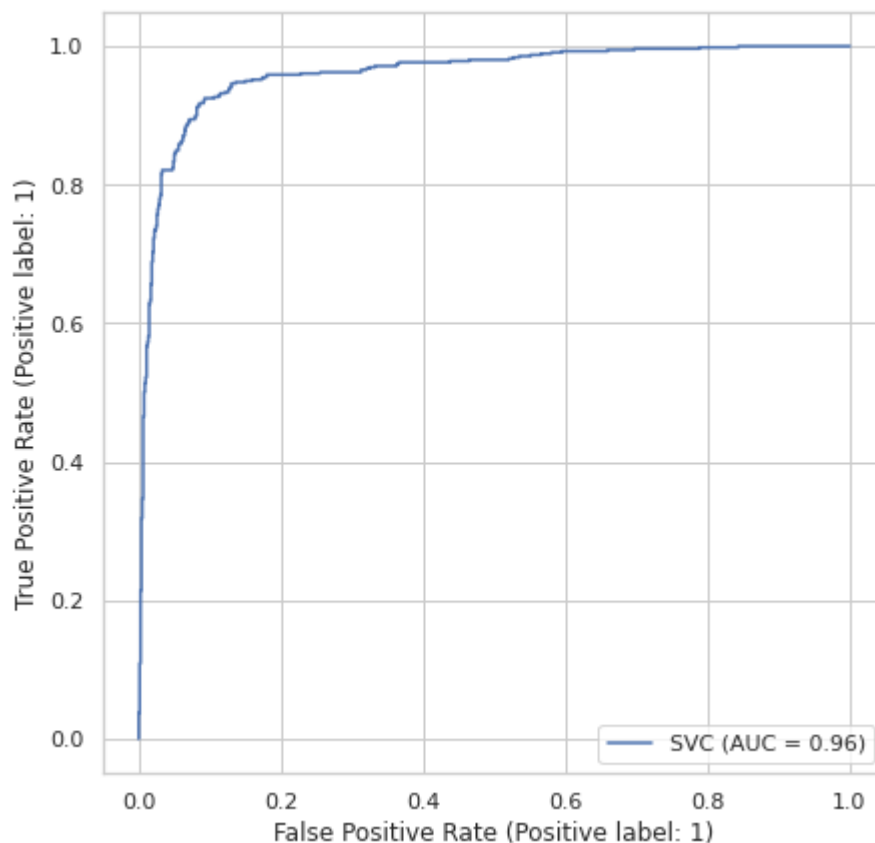
Finally we got 89% around the score, Overall the model received a very good score, demonstrating that SVM performs well when applied to identifying spam.

Accuracy: 0.89 (+/- 0.07)

5.2.2. Plot the ROC Curve and AUC

In sklearn, there are already self-contained functions to complete the ROC curve, which is very convenient.

```
from sklearn.metrics import plot_roc_curve  
plot_roc_curve(svc, X_test, y_test)  
plt.show()
```



According to the graph, we can see that the curve is quite far from the median score of 0.5, and we get a value of $1 > 0.96 > 0.5$ for the AUC. The classifier with a larger AUC works better.

AUC means that when a positive sample and a negative sample are picked at random, the probability of ranking this positive sample ahead of the negative sample according to the score calculated by the current classifier.

5.2.3. Confusion Matrix

We then used the Confusion Matrix to evaluate the model.

```
cnf_matrix = confusion_matrix(y_test, y_pred=svc.predict(X_test))
```

```
def plot_confusion_matrix(cm, classes, title='Confusion matrix',
cmap='viridis'):
    plt.figure(figsize=(8,8))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, fontsize=28)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
```

```

plt.xticks(tick_marks, classes, rotation=45, fontsize=12)
plt.yticks(tick_marks, classes, fontsize=12)

thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j],
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black",
             fontsize=24)

plt.tight_layout()
plt.ylabel('True label', fontsize=18)
plt.xlabel('Predicted label', fontsize=18)
plt.show()

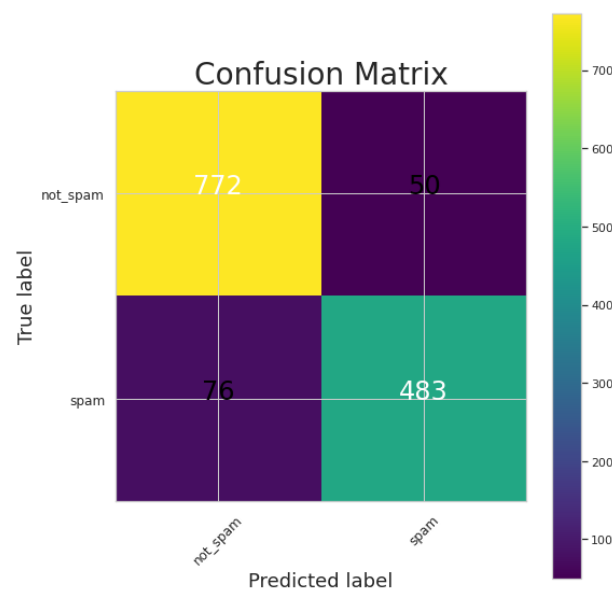
```

```

plot_confusion_matrix(cnf_matrix, classes=('not_spam', 'spam'),
title='Confusion Matrix')

```

Finally, we get the Confusion Matrix, and from the graph, it appears that emails that should not be spam are identified as spam in a somewhat large number.



5.3. Results and discussion

In general, the performance of SVM and logistic regression classifiers is not too different, and overall, one may prefer to use whichever model is more efficient.

6. Machine learning model 3 (MLP Neural Network)

6.1. Summary of the approach

Neural networks Common machine learning techniques used to design neural network application scenarios include supervised and unsupervised learning, classification, regression, pattern recognition and clustering. In this subsection, I use the Neural Networks Model to make judgments about spam.

I am going to use Keras to build a neural network model. For higher accuracy, I will build a seven-layer neural network (not counting the input and output layers). After the model has been compiled, the model accuracy calculation is completed with a test set. To better calculate the average accuracy, I introduced a 10-fold cross-validation and used the pipeline in the sklearn library to perform a wrapper.

6.2. Preparing data

Firstly, we should again re-prepare the data in case the previous experiment had an impact on the content of the variables.

```
spambase =  
pd.read_csv('https://datahub.io/machine-learning/spambase/r/spambase.csv'  
)  
  
spambase.drop(columns=["word_freq_george", "word_freq_650"], axis =1,  
inplace=True)  
  
# Based on the features, we select the input elements and targets.  
X = spambase.iloc[:,0:55]  
y = spambase.iloc[:,55]  
  
# split the data into train set and test set  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,  
random_state=0)
```

Standardisation before data entry can be very effective in improving the speed and effectiveness of convergence. **As the classification in our dataset already uses numbers as classification labels, we only need to normalise the training data.**

Because our activation function is sigmoid, the gradient of its maximum interval is around 0. When the input value is large or small, the change in sigmoid or tanh is basically flat, which means that when gradient descent is performed for optimisation, the gradient will converge to 0, resulting in a slow optimisation speed.

```
scaler = StandardScaler().fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train), index=y_train)
X_test = pd.DataFrame(scaler.transform(X_test), index=y_test)
```

6.3. Model training and evaluation

After preparing the data, we are ready to build the model.

We are going to build a seven-layer neural network, choosing seven layers instead of one for higher accuracy, and choosing a higher number of iterations.

In order to make the model generalise better, we added a "dropout" layer between the layers and set the value to 0.5. Half of the hidden elements are randomly discarded as the neurons of the neural network are passed backwards.

Because it is a dichotomous problem, a binary cross-entropy function is used as the loss function, and the evaluation criterion is still accuracy.

```
model = Sequential()
model.add(Dense(55, input_dim=55, activation='relu'))
model.add(Dense(55, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(55, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(55, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(55, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(55, activation='relu'))
model.add(Dropout(0.5))
```



```

model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='rmsprop',
metrics='accuracy')
model.fit(X_train, y_train, batch_size=1024, epochs=1000, verbose=0)
# if you want to see the verbose log, you can change the
"verbose='auto'"
model.summary()

```

After the model has been compiled, we print out some information about the model and can see some specific parameters.

```

Model: "sequential"

```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 55)	3080
dense_1 (Dense)	(None, 55)	3080
dropout (Dropout)	(None, 55)	0
dense_2 (Dense)	(None, 55)	3080
dropout_1 (Dropout)	(None, 55)	0
dense_3 (Dense)	(None, 55)	3080
dropout_2 (Dropout)	(None, 55)	0
dense_4 (Dense)	(None, 55)	3080
dropout_3 (Dropout)	(None, 55)	0
dense_5 (Dense)	(None, 55)	3080
dropout_4 (Dropout)	(None, 55)	0
dense_6 (Dense)	(None, 55)	3080
dropout_5 (Dropout)	(None, 55)	0
dense_7 (Dense)	(None, 55)	3080
dropout_6 (Dropout)	(None, 55)	0
dense_8 (Dense)	(None, 1)	56

```

Total params: 24,696
Trainable params: 24,696
Non-trainable params: 0

```

Next, we need to use our test set to verify the accuracy of our model. And we will get the loss value and metrics values for the model in test mode.

```

results = model.evaluate(X_test, y_test, batch_size=512)
print("test loss, test acc:", results)

```

```

| results = model.evaluate(X_test, y_test, batch_size=512)
| print("test loss, test acc:", results)

```

```

3/3 [=====] - 0s 4ms/step - loss: 5.4397 - accuracy: 0.9283
test loss, test acc: [5.439732551574707, 0.9283128380775452]

```

6.4. k-fold crossValidation

We will use k-fold cross-validation in sklearn to evaluate the model. We must use the Keras model with scikit-learn to use the KerasClassifier wrapper. This class accepts a function to create and return our neural network model.

Let us start by defining the function that creates the baseline model. Our model will have a fully connected hidden layer with the same number of neurons as the input variables. This is a good default starting point when creating a neural network.

The weights are initialised using small Gaussian random numbers. A Rectifier activation function is used. The output layer contains one neuron to make predictions. As our task is a binary classification problem, the model ends up using a sigmoid activation function to produce a probabilistic output of 0 or 1.

Finally, we use a logarithmic loss function (binary_crossentropy) during training, which is the preferred loss function for binary classification problems. The model also uses an efficient Adam optimisation algorithm for gradient descent and collects accuracy metrics as the model is trained.

We provide the KerasClassifier with the number of training epochs with appropriate default settings. We turn off the westward output because the model will be created ten times to perform 10-fold cross-validation.

The best way to do this is to train a normalisation process on the training data during the cross-validation run and use the trained normalisation to prepare the 'hidden' test folds. This makes normalisation a step in preparing the model during cross-validation, and it prevents the algorithm from learning about the 'invisible' data during the evaluation process. This knowledge might be passed on from the data preparation scheme, such as a more precise distribution.

Furthermore, for a more intuitive representation, we can do this using a pipeline from the sklearn library, a wrapper that executes one or more models in the pass of the cross-validation process. Here we can use StandardScaler to define a pipeline followed by our neural network model.

```
# binary classification NN model
def create_BCNN():
    # create model
    model = Sequential()
    model.add(Dense(55, input_dim=55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(55, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
```

```

model.compile(loss='binary_crossentropy', optimizer='rmsprop',
metrics='accuracy')
return model

```

```

# Cross Val score
from sklearn.model_selection import cross_val_score
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import StratifiedKFold

# evaluate baseline model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_BCNN,
epochs=500, batch_size=1024, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True)
results = cross_val_score(pipeline, X, y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100,
results.std()*100))

```

It can be seen that after 10-fold cross-validation, the average accuracy of the model can be obtained to be about 92% and the standard deviation is also small.

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:

```

```

KerasClassifier is deprecated, use Sci-Keras (https://github.

```

```

WARNING:tensorflow:5 out of the last 7 calls to <function Mod

```

```

WARNING:tensorflow:6 out of the last 8 calls to <function Mod

```

```

Standardized: 92.70% (2.11%)

```

6.5. Results and discussion

From the multi-layer perceptron comparing the previous two methods, the accuracy is the highest, although it also consumes the most resources. Perhaps the difference in accuracy, because it is only a simple binary classification problem, is not particularly large. Furthermore, compared to the first two models, the model building and understanding is considerably more.

7. Results comparison across the models built

At the final stage, we should compare the three models, and the best comparison is to take out the data and list them together for comparison.

Re-prepare data:

```
spambase =
pd.read_csv('https://datahub.io/machine-learning/spambase/r/spambase.csv')

spambase.drop(columns=["word_freq_george", "word_freq_650"], axis =1,
inplace=True)

# Based on the features, we select the input elements and targets.
X = spambase.iloc[:,0:55]
y = spambase.iloc[:,55]

# split the data into train set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
random_state = 0)

scaler = StandardScaler().fit(X_train)
X_train = pd.DataFrame(scaler.transform(X_train), index=y_train)
X_test = pd.DataFrame(scaler.transform(X_test), index=y_test)
```

Next, we create a new dataframe and store all the various data that we have finally calculated in it. Three separate models were used to predict to calculate accuracy, and their Accuracy, Precision, Recall and F1 values were calculated.

```
results = pd.DataFrame()
time_consumption = []

start = time.time()
results['logical_regression'] = LR.predict(X_test)
end = time.time()
time_consumption.append(end - start)

start = time.time()
results['SVC'] = svc.predict(X_test)
end = time.time()
time_consumption.append(end - start)
```

```

start = time.time()
results['Neural_Network'] = np.around(model.predict(X_test)).reshape(-1)
end = time.time()
time_consumption.append(end - start)

evaluations = pd.DataFrame()
from sklearn.metrics import *
for i in range(0, results.shape[1]):
    evaluation = {"precision": precision_score(test_y, results.iloc[:,i]),
                  "recall": recall_score(test_y, results.iloc[:,i]),
                  "accuracy": accuracy_score(test_y, results.iloc[:,i]),
                  "f1": f1_score(test_y, results.iloc[:,i])}
    evaluations = evaluations.append(evaluation, ignore_index=True)
evaluations.rename(index=dict(zip(range(0, evaluations.shape[0]),
                                   list(results.columns))), inplace=True)

evaluations.insert(0, 'time/s', time_consumption)

evaluations

```

Finally we can see the comparison data stored in the dataframe.

evaluations

	time/s	precision	recall	accuracy	f1
logical_regression	0.009497	0.911877	0.851521	0.906589	0.880666
SVC	0.061056	0.906191	0.864043	0.908762	0.884615
Neural_Network	0.235512	0.945098	0.862254	0.923968	0.901777

8. Conclusion, recommendations and future work

Overall, the neural network is the best compared to the other two models. However, probably due to the simplicity of the problem or the fact that the data set is still not huge enough, the difference in accuracy is no more than 2%. The neural network is more GPU and other resource intensive than the other two models, but has more options and room for optimisation.

In future research, I can try to use more neural network models to train the dataset and achieve higher accuracy while avoiding overfitting.

9. References

- (1) “A Classification and Regression Tree (CART) Algorithm.” 2021. Analytics Steps.
<https://www.analyticssteps.com/blogs/classification-and-regression-tree-cart-algorithm>.
- (2) Hopkins, Mark, Erik Reeber, George Forman, and Jaap Suermondt. n.d. “Spambase - Dataset.” DataHub. Accessed February 7, 2022.
<https://datahub.io/machine-learning/spambase#data>.
- (3) “Logistic regression.” n.d. Wikipedia. Accessed February 7, 2022.
https://en.wikipedia.org/wiki/Logistic_regression.
- (4) “Multidimensional scaling.” n.d. Wikipedia. Accessed February 9, 2022.
https://en.wikipedia.org/wiki/Multidimensional_scaling.
- (5) “parrr/dtreviz: A python library for decision tree visualization and model interpretation.” n.d. GitHub. Accessed February 8, 2022. <https://github.com/parrr/dtreviz>.
- (6) “Principal component analysis.” n.d. Wikipedia. Accessed February 8, 2022.
https://en.wikipedia.org/wiki/Principal_component_analysis.
- (7) “Scatter matrix.” n.d. Wikipedia. Accessed February 8, 2022.
https://en.wikipedia.org/wiki/Scatter_matrix.
- (8) SethEBaldwin. n.d. “mdscuda.” GitHub. Accessed 2 9, 2022.
<https://github.com/SethEBaldwin/mdscuda>.
- (9) “Spambase Data Set.” n.d. UCI Machine Learning Repository. Accessed February 8, 2022.
<http://archive.ics.uci.edu/ml/datasets/Spambase>.