ECON3203 Group Assignment Report (Group 32)

Bill Yang - z5394798,
Garret Lin - z5359844
Adelyn Chan - z5363867
Amanda Chang - z5361911
Cindy Zhang - z5363808

1. Intro & Executive Summary

The purpose of this report is to develop a strong predictive model for ATM cash demand for banks to optimise their cash management with. This is done by employing various supervised machine learning model techniques on a provided training dataset to develop the best model when used on a separate dataset reserved for testing. The prediction will be made upon the response variable 'Withdraw' (the amount withdrawn), based on a variety of combinations of the other variables - 'Shops', 'ATMs', 'Downtown', 'Weekday', 'Center' and 'High', and the 'best prediction' will be defined as one with the lowest Test Error (or Mean Squared Error) between the response variable - calculated from the Test Data covariates - and its actual value within the Test Dataset. The final best prediction will subsequently determine the final best model used.

2. Methodology:

2.1 Preparation: Exploratory Data Analysis

As with all analytical procedures, we must first conduct relevant EDA to ensure that our datasets are suitable for training and then testing. The following sections are to provide a broad explanation on how EDA was conducted prior to selecting the best variables and models.

2.1.0 Removing NULL data

Upon inspection of both CSV files containing the data to be used, we observed the presence of NULL values within the datasets. To remove these, the dropna() function was employed. It is important that any NULL data points are removed as these values may lead to unintentional errors within the model. *fig.0* shows the code used to do this.

```
import numpy as np

ATM_train = pd.read_csv("ATM_training.csv")
ATM_train = ATM_train.dropna()

ATM_test = pd.read_csv("ATM_test.csv")
ATM_test = ATM_test.dropna()
```

fig.0 Removing NULL Values

2.1.1 Variable Selection Problem

The first step in the variable selection process was to investigate which of the covariates could potentially be 'dropped' with relation to the final value of *'Withdraw'* (as 6 variables are present, there exists a total of 2^6 subsets). To do this, **Backward-Stepwise Selection** was used, which works by comparing the AIC of different models with a single variable removed. The first step of this process is demonstrated in *fig.1*. The code essentially removes one of the covariates from

the standard linear regression equation (including all original variables), calculates the AIC of the new model and repeats this process for all covariates. The output of this code revealed the AIC of a model when a certain variable specified was removed. The AIC of 'None' (i.e. removing no variables) can be seen to be the lowest. Surprisingly this meant that none of the covariates within the initial data should be removed in order to minimise AIC, and as this happened on the first step, only one iteration of this code was necessary.

```
import statsmodels.formula.api as smf
aics = {'None':[],'Shops':[], 'ATMs':[], 'Downtown':[], 'Weekday':[], 'Center':[],'High':[]}

formula = 'Withdraw ~ Shops + ATMs + Downtown + Weekday + Center + High'
model = smf.ols(formula, ATM_train)
results = model.fit()
aics["None"].append(results.aic)

for predictor in ['Shops', 'ATMs', 'Downtown', 'Weekday', 'Center','High']:
  array = ['Shops', 'ATMs', 'Downtown', 'Weekday', 'Center','High']
  formula = 'Withdraw ~ '
  array.remove(predictor)
  count = 0
  for pred in array:
    if count == 0:
      formula += ' ' + pred
      count += 1
    else:
      formula += ' + ' + pred
  model = smf.ols(formula, ATM_train)
  results = model.fit()
  aics[predictor].append(results.aic)
for val in sorted(aics.items(), key=lambda x:x[1]):
  print(val)
```

```
('None', [102773.75332957253])
('High', [103439.54001957676])
('Downtown', [104375.47924993062])
('Weekday', [110175.78819794345])
('ATMs', [111989.15389449417])
('Shops', [112424.65910466018])
('Center', [115218.43711395137])
```

fig.1 AIC Analysis for Model Selection

2.1.2 Interaction Effects and Significance

To improve model accuracy for linear regression and other models, it's important that any highly significant nonlinear interactions between the current covariates is introduced. However, one point to note is that there are a huge number of combinations of non-linear interactions between these covariates, depending on the degree of the model equation. Thankfully, we do not need to go through every single one of these combinations, by following the steps below.

To find the best interactions:

1. We first find every single combination of degree '2'
2. If degree '2' produced any significant variables**, we go up by one degree - '3'

3. We keep investigating every combination up to one degree, until there are no significant interactions between, say for example, any variables in degree '4'. At this point, we use the significant interactions found in the degree before this.

***Significant variables were defined as those with a P-value of < 0.001 (very strong evidence) during EDA.*

```
from sklearn.preprocessing import PolynomialFeatures
X_train = ATM_train[ATM_train.columns.drop('Withdraw')]
y_train = ATM_train[['Withdraw']]
poly = PolynomialFeatures(2)
output_nparray = poly.fit_transform(X_train)
dc = poly.get_feature_names(X_train.columns)
X_train = pd.DataFrame(output_nparray, columns = dc)
X_train = X_train.drop('1', axis = 1)
X_train.head()
```

```
import statsmodels.api as sm
x_with_intercept = sm.add_constant(X_train, prepend=True)
model = sm.OLS(y_train, x_with_intercept)
results = model.fit()  # perform linear regression

print(results.summary())
```

fig.2 Code to Generate All Interaction Combinations based on Degree

Degree '2' Results:

- Degree 2 produced two significant interactions (as highlighted in *fig.3*)

	coef	std err	t	P>|t|	[0.025	0.975]
const	17.1552	0.374	45.903	0.000	16.423	17.888
Shops	2.1344	0.477	4.478	0.000	1.200	3.069
ATMs	-0.9637	0.032	-30.457	0.000	-1.026	-0.902
Downtown	-9.6142	8.757	-1.098	0.272	-26.778	7.550
Weekday	-0.2206	0.060	-3.706	0.000	-0.337	-0.104
Center	6.4738	0.090	71.666	0.000	6.297	6.651
High	0.4017	0.060	6.641	0.000	0.283	0.520
Shops^2	0.3136	0.175	1.788	0.074	-0.030	0.657
Shops ATMs	-0.0314	0.025	-1.272	0.203	-0.080	0.017
Shops Downtown	3.0778	3.177	0.969	0.333	-3.149	9.304
Shops Weekday	-0.3191	0.109	-2.939	0.003	-0.532	-0.106
Shops Center	0.6491	0.162	4.011	0.000	0.332	0.966
Shops High	0.1746	0.108	1.617	0.106	-0.037	0.386
ATMs^2	-0.0018	0.002	-1.039	0.299	-0.005	0.002
ATMs Downtown	0.2991	0.224	1.337	0.181	-0.139	0.738
ATMs Weekday	0.0183	0.010	1.746	0.081	-0.002	0.039
ATMs Center	-0.0084	0.015	-0.540	0.589	-0.039	0.022
ATMs High	-0.0051	0.010	-0.498	0.619	-0.025	0.015
Downtown^2	-9.6142	8.757	-1.098	0.272	-26.778	7.550
Downtown Weekday	0.8639	0.981	0.881	0.379	-1.059	2.787
Downtown Center	-0.2855	1.461	-0.195	0.845	-3.148	2.577
Downtown High	-1.5344	0.976	-1.573	0.116	-3.447	0.378
Weekday^2	-0.2206	0.060	-3.706	0.000	-0.337	-0.104
Weekday Center	-14.4095	0.061	-235.740	0.000	-14.529	-14.290
Weekday High	0.0433	0.041	1.065	0.287	-0.036	0.123
Center^2	6.4738	0.090	71.666	0.000	6.297	6.651
Center High	-0.2065	0.060	-3.442	0.001	-0.324	-0.089
High^2	0.4017	0.060	6.641	0.000	0.283	0.520

fig.3 Degree 2 Interactions Linear Regression Model

Degree ‘3’ Results:

- Degree 3 produced only one significant interaction (as highlighted in *fig.4*)

```
x_with_intercept = sm.add_constant(X_train, prepend=True)
model = sm.OLS(y_train, x_with_intercept)
results = model.fit()  # perform linear regression

print(results.summary())
```

Shops ATMs Weekday	0.0540	0.022	1.567	0.117	-0.009	0.077
Shops ATMs Center	-0.0212	0.032	-0.663	0.508	-0.084	0.041
Shops ATMs High	0.0131	0.022	0.603	0.546	-0.029	0.056
Shops Downtown^2	33.5463	30.379	1.104	0.269	-25.998	93.090
Shops Downtown Weekday	-1.9088	2.781	-0.686	0.493	-7.360	3.542
Shops Downtown Center	0.1945	4.177	0.047	0.963	-7.993	8.382
Shops Downtown High	-6.2531	2.787	-2.244	0.025	-11.715	-0.791
Shops Weekday^2	-0.1731	0.208	-0.833	0.405	-0.580	0.234
Shops Weekday Center	-2.1893	0.144	-15.163	0.000	-2.472	-1.906
Shops Weekday High	-0.0553	0.095	-0.581	0.561	-0.242	0.131
Shops Center^2	0.8483	0.308	2.752	0.006	0.244	1.452
Shops Center High	-0.0781	0.140	-0.558	0.577	-0.353	0.196
Shops High^2	-0.2680	0.207	-1.293	0.196	-0.674	0.138
ATMs^3	3.384e-05	0.000	0.164	0.870	-0.000	0.000
ATMs^2 Downtown	-0.0258	0.032	-0.808	0.419	-0.088	0.037
ATMs^2 Weekday	-0.0019	0.002	-1.213	0.225	-0.005	0.001
ATMs^2 Center	0.0030	0.002	1.361	0.173	-0.001	0.007
ATMs^2 High	-0.0022	0.002	-1.483	0.138	-0.005	0.001
ATMs Downtown^2	2.0789	1.769	1.175	0.240	-1.388	5.545
ATMs Downtown Weekday	-0.2815	0.197	-1.428	0.153	-0.668	0.105

fig.4 Degree 3 Interactions Linear Regression Model

Degree ‘4’ Results:

- Degree 4 produced no significant interactions

Bringing these results together, since all non-linear combinations of degree ‘3’ produced only one significant non-linear interaction, this will be the only chosen interaction effect that will be used within our models. By the **hierarchy principle**, however, we must also include any associated effects of this interaction term. Hence our final model equation should look like the following equation:

$$Withdraw = \beta_1 Shops + \beta_2 ATMs + \beta_3 Downtown + \beta_4 Weekday + \beta_5 Center + \beta_6 High + \beta_7 Shops \times Center + \beta_8 Shops \times Weekday + \beta_9 Weekday \times Center + \beta_{10} Shops \times Weekday \times Center$$

fig.5 Determined Model Equation

2.2 Model: Linear Regression

2.2.0 Importing Datasets

```
#Read the training dataset into a dataframe, and drop null values
ATM_train = pd.read_csv("ATM_training.csv")
ATM_train = ATM_train.dropna()
ATM_train.head()
```

```
#Read the testing dataset into a dataframe, and drop null values
ATM_test = pd.read_csv("ATM_test.csv")
ATM_test = ATM_test.dropna()
ATM_test.head()
```

fig.6 Importing Datasets

2.2.1 Setting Predictors and Response Variable

We need to make sure we add the interaction effects predetermined in Section 2.1.2

```
ATM_train['Shops:Center'] = ATM_train['Shops']*ATM_train['Center']
ATM_train['Shops:Weekday'] = ATM_train['Shops']*ATM_train['Weekday']
ATM_train['Weekday:Center'] = ATM_train['Weekday']*ATM_train['Center']
ATM_train['Shops:Weekday:Center'] = ATM_train['Shops']*ATM_train['Weekday']*ATM_train['Center']
```

```
ATM_test['Shops:Center'] = ATM_test['Shops']*ATM_test['Center']
ATM_test['Shops:Weekday'] = ATM_test['Shops']*ATM_test['Weekday']
ATM_test['Weekday:Center'] = ATM_test['Weekday']*ATM_test['Center']
ATM_test['Shops:Weekday:Center'] = ATM_test['Shops']*ATM_test['Weekday']*ATM_test['Center']
```

```
#Define the withdraw variable as the response variable and the rest as the independent variables
response = ['Withdraw']
predictors = [x for x in list(ATM_train.columns) if x not in response]
X_train = ATM_train[predictors]
y_train = ATM_train[response]
X_test = ATM_test[predictors]
y_test = ATM_test[response]
```

fig.7 Initiating Predictors and Response Variable

2.2.2 Computing The Multiple Linear Regression Model

The formula used for the Multiple Linear Regression Model was previously determined in Section 2.1.2 in *fig.5* to include the interaction effect ‘Shops * Weekdays * Center’. We used the StatsModel Python Library, which computed the model using OLS on the training dataset (ATM_training.csv) as shown below in *fig.8*:

```python
import statsmodels.api as sm
formula = 'Withdraw ~ Shops + ATMs + Downtown  + Weekday + Center + High + Shops*Center +
Shops*Weekday + Weekday*Center + Shops*Weekday*Center'

model = smf.ols(formula, ATM_train)

results = model.fit()

print(results.summary())
```

fig.8 Computing Linear Regression using StatsModels

2.2.3 Model Results

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               Withdraw   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 4.460e+06
Date:                Fri, 11 Nov 2022   Prob (F-statistic):               0.00
Time:                        22:39:59   Log-Likelihood:                -18348.
No. Observations:               22000   AIC:                         3.672e+04
Df Residuals:                   21989   BIC:                         3.681e+04
Df Model:                          10
Covariance Type:            nonrobust
================================================================================================
                                   coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------------------
Intercept                        9.7600      0.027    359.309      0.000       9.707       9.813
Shops                           10.7660      0.022    490.397      0.000      10.723      10.809
ATMs                            -1.0004      0.002   -475.688      0.000      -1.005      -0.996
Downtown                       -36.3879      0.198   -184.014      0.000     -36.776     -36.000
Weekday                         -2.0227      0.018   -113.805      0.000      -2.058      -1.988
Center                           2.9908      0.045     66.134      0.000       2.902       3.079
High                             0.9990      0.008    122.009      0.000       0.983       1.015
Shops:Center                     1.9974      0.005    366.123      0.000       1.987       2.008
Shops:Weekday                    0.0027      0.002      1.279      0.201      -0.001       0.007
Weekday:Center                  -0.0337      0.055     -0.618      0.537      -0.141       0.073
Shops:Weekday:Center            -1.9926      0.007   -304.672      0.000      -2.005      -1.980
==============================================================================
Omnibus:                       44.769   Durbin-Watson:                   1.989
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               55.529
Skew:                           0.013   Prob(JB):                     8.75e-13
Kurtosis:                       3.245   Cond. No.                         728.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

fig.9 Linear Regression Model Summary

The model produced was an extremely strong representation of the Training Dataset due to the R-Squared of 1.000 - a virtually perfect fit. This proved that the independent variables, along with the strong interaction covariates found in our EDA in Section 2.1, explained the response variable of ‘Withdraw’ extremely well. There is also no apparent **multicollinearity** within this

model due to the relatively low Cond. Number, as well as a relatively low AIC proving model suitability.

2.2.4 Test Prediction and Test Error (MSE)

Now that we have an extremely strong model for Linear Regression, the most important results are to look at how this model performs when used to predict the response variable 'Withdraw' on the Testing Dataset (ATM_Test.csv). To do this we simply used the model.predict() function with StatsModels:

```
y_pred = results.predict(X_test)
```

Note: X_test was previously defined above

These predicted y-values are then compared with the actual values within the 'Withdraw' column of the Testing Dataset, and the Mean Squared Error is then calculated by using the built in MSE function from the Python Library 'sklearn', as shown below in *fig.10*:

```
from sklearn.metrics import mean_squared_error

mean_squared_error(ATM_test['Withdraw'], y_pred).round(4)
```

```
0.3089
```

fig.10 Linear Regression Test Error

For our Linear Regression Model, a Test Error of 0.3089 was obtained.

2.3 Model: Lasso Regression

2.3.0 Setting Up Datasets and Variables

Datasets to be used for training/testing with Lasso Regression are the same as the ones used in Section 2.2.0, but modified to include the interaction in Section 2.2.1 after EDA. Predictors and response variables are also the same as shown in Section 2.2.1.

2.3.1 Regularisation: Standardising Numerical Predictors

As Lasso Regression is a regularisation technique (L1 Regularisation), it is important to first standardise the numerical predictors by subtracting the mean and then dividing by the standard deviations.

```
numerical_predictors=['Shops', 'ATMs', 'Shops:Center', 'Shops:Weekday',
'Shops:Weekday:Center']
mu=ATM_train[numerical_predictors].mean() # mean for each feature
sigma=ATM_train[numerical_predictors].std() # std for each feature
ATM_train[numerical_predictors]=(ATM_train[numerical_predictors]-mu)/sigma
ATM_test[numerical_predictors]=(ATM_test[numerical_predictors]-mu)/sigma
```

fig.11 Standardising Predictors using Pandas Library

2.3.2 Computing The Lasso Regression Model

To compute our Lasso Regression Model using our set predictors and response variables, we used the LassoCV function from the Python Library 'sklearn.linear_model'. LassoCV is used over regular Lasso because it has built in CV-based model selection for a tuning parameter 'alpha', i.e. it will automatically find the optimal alpha for the penalty term. We generated our model using LassoCV with a cross-validation score of 10 and then fitted this model to our predictors and response variable using StatsModels:

```
from sklearn.linear_model import LassoCV

lasso = LassoCV(cv=10)
lasso.fit(ATM_train[predictors], np.ravel(ATM_train[response]))
```

```
print("LASSO Lambda: {0}".format(lasso.alpha_))
```

```
LASSO Lambda: 0.024742158373629215
```

fig.12 Using LassoCV to Fit Training Model

The optimal lambda selected by CV is 0.0247, as shown above in *fig.12.*

2.3.3 Model Results

The predictors selected by the optimal Lasso model are shown in *fig.13* below. It's clear that Lasso has reduced some predictors such as Downtown to 0.

```
pd.DataFrame(lasso.coef_.round(3),index = predictors).T
```

	Shops	ATMs	Downtown	Weekday	Center	High	Shops:Center	Shops:Weekday	Weekday:Center	Shops:Weekday:Center
0	27.588	-3.592	-0.0	-1.919	2.094	0.885	5.265	-0.0	0.0	-4.266

fig.13 Lasso Predictors

2.3.4 Test Prediction and Test Error (MSE)

We now look at how this model performs when used to predict the response variable 'Withdraw' on the Testing Dataset (ATM_Test.csv). To do this we simply used the model.predict() function with StatsModels. The mean squared Test Error was then determined using sklearn:

```
y_pred = lasso.predict(X_test)
test_error = mean_squared_error(ATM_test['Withdraw'], y_pred).round(4)
print(test_error)
```

```
0.9816
```

fig.14 Lasso Regression Test Error

For our Lasso Regression Model, a Test Error of 0.9816 was obtained.

2.4 Model: Ridge Regression

2.4.0 Setting Up Datasets and Variables

Datasets to be used for training/testing with a Ridge Regression model are the same as the ones used in Section 2.2.0, modified to include the interaction in Section 2.2.1 after EDA. Predictors and response variables are also the same as shown in Section 2.2.1.

2.4.1 Regularisation: Standardising Numerical Predictors

As Lasso Regression is a regularisation technique (L2 Regularisation), it is important to first standardise the numerical predictors by subtracting the mean and then dividing by the standard deviations. This process was done in Section 2.3.1 as Lasso Regression also required this.

2.4.2 Computing The Ridge Regression Model

To compute our Ridge Regression Model using our set predictors and response variables, we used the RidgeCV function from the Python Library 'sklearn.linear_model'. However, unlike Lasso, we had to manually specify a grid on penalty values as shown in *fig.15* below. The optimal Lambda selected by CV=5 is 0.0005.

```
from sklearn.linear_model import RidgeCV

alphas = np.exp(np.linspace(-10,20,500))
ridge_cv = RidgeCV(alphas=alphas, cv=5)
ridge_cv.fit(ATM_train[predictors], np.ravel(ATM_train[response]))
```

```
print("Ridge Lambda: {0}".format(ridge_cv.alpha_))
```

```
Ridge Lambda: 0.0005340238055689655
```

fig.15 Using RidgeCV to Fit Training Model

2.4.3 Model Results

The predictors selected by the optimal Ridge Regression model are shown in *fig.16* below. Unlike Lasso, Ridge does not reduce predictors down to 0:

```
pd.DataFrame(ridge.coef_.round(3),index = predictors).T
```

	Shops	ATMs	Downtown	Weekday	Center	High	Shops:Center	Shops:Weekday	Weekday:Center	Shops:Weekday:Center
0	44.34	-3.675	-36.384	-2.023	2.991	0.999	5.156	0.013	-0.034	-4.394

fig.16 Ridge Predictors

2.4.4 Test Prediction and Test Error (MSE)

We now look at how this model performs when used to predict the response variable ‘Withdraw’ on the Testing Dataset (ATM_Test.csv). To do this we simply used the model.predict() function with StatsModels. The mean squared Test Error was then determined using sklearn:

```
y_pred = ridge.predict(X_test)
test_error = mean_squared_error(ATM_test['Withdraw'], y_pred).round(4)
print(test_error)
```

```
0.3089
```

fig.17 Ridge Regression Test Error

For our Ridge Regression Model, a Test Error of 0.9816 was obtained.

2.5 Model: Elastic Net CV

2.5.0 Setting Up Datasets and Variables

Datasets to be used for training/testing with a Elastic Net CV model are the same as the ones used in Section 2.2.0, modified to include the interaction in Section 2.2.1 after EDA. Predictors and response variables are also the same as shown in Section 2.2.1.

2.5.1 Regularisation: Standardising Numerical Predictors

Standardising Numerical Predictors for Elastic Net CV was done in 2.3.1 for Lasso Regression and carried over for this process.

2.5.2 Computing The Elastic Net CV Model

To compute our Elastic Net CV Model using our set predictors and response variables, we used the ElasticNetCV function from the Python Library 'sklearn.linear_model'. Unlike Lasso and Ridge however, we needed to specify a grid of values for the weight on lasso and ridge penalties as shown in *fig.18* below.

```
from sklearn.linear_model import ElasticNetCV
enet_cv = ElasticNetCV(l1_ratio=[0.01,0.05,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,0.95,0.99], cv=5)
enet_cv.fit(ATM_train[predictors],np.ravel(ATM_train[response]))
```

fig.18 Using ElasticNetCV to Fit Training Model

2.5.3 Test Prediction and Test Error (MSE)

Following the same procedures to determine the test error as those in Sections 2.3.3 and 2.4.3 for Lasso and Ridge, we used the fitted ElasticNetCV model to predict our response variable and 'sklearn' to calculate the MSE:

```
y_pred = enet_cv.predict(X_test)
test_error = mean_squared_error(ATM_test['Withdraw'], y_pred).round(4)
print(test_error)
```

```
0.9858
```

fig.19 Elastic Net CV Test Error

For our Elastic Net CV Model, a Test Error of 0.9858 was obtained.

2.6 Model: KNN Classifier

2.6.0 Setting Up Libraries

The KNN Classifier model to be used is provided by the Python library 'sklearn'. Libraries used are shown below, with a 10-fold cross-validation to be used:

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
kf=KFold(10)
```

fig.20 Importing KNN Libraries

2.6.1 Setting Up Datasets and Variables

Datasets to be used for training/testing with a KNN Classifiers model are the same as the ones used in Section 2.2.0, modified to include the interaction in Section 2.2.1 after EDA. Predictors and response variables are also the same as shown in Section 2.2.1.

2.6.2 Computing The KNN Classifier Model

To compute the KNN Model, a function was developed [ref. Tute 8 of ECON3203] with parameters set as the predictors and response variable. The function runs a for-loop that iterates KNN with 1 to 51 neighbours, and then determines the best number of neighbours using the cross-validated score. This best number of neighbours is then used to fit a KNN model, producing the optimal Test Error and Cross Validation Error which are the two values returned.

Note: the metric/method used for this function was Mahalanobis Distance. Surprisingly also, the best KNN error was produced when the vanilla dataset for both ATM_training and ATM_test were used, with no interaction effects:

```
ATM_train = pd.read_csv("ATM_training.csv")
ATM_train = ATM_train.dropna()

ATM_test = pd.read_csv("ATM_test.csv")
ATM_test = ATM_train.dropna()
```

fig.21 Vanilla Datasets Used for KNN

```
def knn_test(predictors, response):

    neighbours=np.arange(1, 51)
    best_score = -np.inf

    for k in neighbours:
        knn = KNeighborsRegressor(n_neighbors = k, metric='mahalanobis', metric_params={'V':
ATM_train[predictors].cov()})
        scores = cross_val_score(knn, ATM_train[predictors], ATM_train[response], cv=10, scoring =
'neg_mean_squared_error')
        # taking the average of scores across 10 folds
        cv_score = np.mean(scores)
        # use the cv score for model selection
        if cv_score >= best_score:
            best_score = cv_score
            best_knn = knn

    knn = best_knn
    # train the selected model with the whole train set
    knn.fit(ATM_train[predictors], ATM_train[response])
    # Predict the test data with the selected and re-estimated model
    predictions = knn.predict(ATM_test[predictors])
    test_mse = mean_squared_error(ATM_test[response], predictions)
    cv_mse= -best_score
    print('Chosen K: {}'.format(knn.n_neighbors))

    return test_mse, cv_mse
```

fig.22 Using KNN with up to 51 Neighbours and CV to Fit Training Model

2.6.3 Test Prediction and Test Error (MSE)

Following the same procedures to determine the test error as per those in the previous sections, we used the fitted optimised KNN model to predict our response variable and 'sklearn' to calculate the MSE. The test_error was the first value of the tuple produced by the function:

```
test_error = knn_test(['Shops', 'ATMs', 'Downtown', 'Weekday', 'Center','High'], 'Withdraw')[0]
```

```
Chosen K: 7
```

```
print(test_error)
```

```
0.25022989535916096
```

fig.23 KNN Classifier Model Test Error

For our KNN Classifier Model, a Test Error of 0.2502 was obtained.

2.7 Model: Neural Network

2.7.0 Setting Up Libraries

The Neural Network model to be used is provided by the machine learning python libraries Keras and Tensorflow. When importing Tensorflow, the RNG seed is preset to ensure we get the same results every time. We then developed a dense NN model.

```
from keras.layers.core import Dense
from keras.models import Sequential
np.random.seed(1)
import tensorflow as tf
tf.random.set_seed(0)
```

fig.24 Importing NN Libraries

2.7.1 Setting Up Datasets and Variables

Datasets to be used for training/testing with a Neural Network model are the same as the ones used in Section 2.2.0, modified to include the interaction in Section 2.2.1 after EDA. Predictors and response variables are also the same as shown in Section 2.2.1.

2.7.2 Pre-processing (Scaling/Standardization)

As Neural Networks normally work best with scaled data, the Python Library sklearn's StandardScaler function was used to standardise the variables used within the dataset.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Fitting the scaler
ATM_train_fit = scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

fig.25 Scaling Datasets

2.7.3 Defining the Feed Forward NN Model

To set up the structure of our Neural Network, we used a Sequential model type (standard). We will first try out one hidden layer with 11 neurons (which is the number of features for the input).

```
#Set up the NN
model = Sequential()
model.add(Dense(11, input_dim=10, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(loss='MSE', optimizer='adam')
```

fig.26 Setting up Neural Network

2.7.4 Computing Model using Adam Optimiser with a Variety of Epochs

As shown in *fig.26*, the first optimizer to train and test our dataset with is the Adam optimization algorithm, which is generally recognised to be one of the best optimizers. To test this however, we must also experimentally determine the best number of **epochs** and the best **batch size** in order to produce the lowest test error.

After running manual tests of epochs of 50, 60, 70, 80, 90 and 100, it was determined that the best epoch size should lie within the range of 60-80 when a batch size of 10 is used, as these values produced MSEs with the lowest error. But how do we do this in an automated way?

A for loop was written to calculate the Test Error of the model for every single integer epoch from range 60 to 80 as shown in *fig.27* below:

```
import keras
#Create an array of epochs to test
epoch_list = range(60,80,1)
#Set up the NN
model = Sequential()
model.add(Dense(11, input_dim=10, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(loss='MSE', optimizer='adam')
#Compute the best test error
test_error = np.inf
for epoch in epoch_list:
  model.fit(X_train, y_train, epochs = epoch, batch_size = 10, verbose = 0)
  error = model.evaluate(X_test, y_test)
  if error < test_error:
    test_error = error
    best_model = model
    best_epoch = epoch
  print(f"Epoch {epoch}: complete, Test Error: {error}")
#Export the best model, so we don't have to run the code again
save_path = './Group_32_Best_NN_Model.h5'
best_model.save(save_path)
print("-----------------------------------------")
print("           FINAL RESULTS                 ")
print("-----------------------------------------")
print('\n MSE of test data using adam optimizer:  ')
print('\nBest epoch no.: ', best_epoch)
print('\n MSE of test data using adam optimizer:  ')
print(test_error)
```

fig.27 For Loop to Calculate the MSE of Predictors Using Different Epoch Sizes

Note: The Python Library ‘keras’ was used within this code to export the best model based on MSE. This is because the for loop that iterates 20 different Epoch Sizes is very computationally expensive. What makes it worse is that it should be run multiple times to mitigate the randomness associated with the stochastic nature of Neural Network architecture. This best model can then be easily reused by simply calling the following code:

```
import keras
## load tensorflow model
save_path = './Group_32_Best_NN_Model.h5'
loaded_model = keras.models.load_model(save_path)
test_error = loaded_model.evaluate(X_test, y_test)
print(test_error)
```

fig.28 Importing and Computing MSE for the Best Exported Model

2.7.5 Model Results

After the loop created in Section 2.7.4 was run, the following output was produced (the code prints the epoch used and test error of every iteration, and the final best epoch and best test error):

```
5/5 [==============================] - 0s 3ms/step - loss: 0.2592
Epoch 68: complete, Test Error: 0.2591647207736969
5/5 [==============================] - 0s 3ms/step - loss: 0.2501
Epoch 69: complete, Test Error: 0.2500969469547272
5/5 [==============================] - 0s 3ms/step - loss: 0.2275
Epoch 70: complete, Test Error: 0.22748659551143646
5/5 [==============================] - 0s 3ms/step - loss: 0.2387
Epoch 71: complete, Test Error: 0.23873983323574066
5/5 [==============================] - 0s 3ms/step - loss: 0.2573
Epoch 72: complete, Test Error: 0.2573460042476654
5/5 [==============================] - 0s 3ms/step - loss: 0.2415
Epoch 73: complete, Test Error: 0.2415093183517456
5/5 [==============================] - 0s 3ms/step - loss: 0.2328
Epoch 74: complete, Test Error: 0.23279327154159546
5/5 [==============================] - 0s 3ms/step - loss: 0.2418
Epoch 75: complete, Test Error: 0.24177581071853638
5/5 [==============================] - 0s 3ms/step - loss: 0.2508
Epoch 76: complete, Test Error: 0.2507748007774353
5/5 [==============================] - 0s 4ms/step - loss: 0.2536
Epoch 77: complete, Test Error: 0.2536466419696808
5/5 [==============================] - 0s 5ms/step - loss: 0.2497
Epoch 78: complete, Test Error: 0.2496698945760727
5/5 [==============================] - 0s 3ms/step - loss: 0.2347
Epoch 79: complete, Test Error: 0.23465220630168915
----------------------------------------
           FINAL RESULTS
----------------------------------------

 MSE of test data using adam optimizer:

Best epoch no.:  70

 MSE of test data using adam optimizer:
0.2275
```

fig.29 Test Errors of Different Epoch Sizes

It is clear that the best test error using the ‘Adam’ optimizer occurred when using an epoch size of 70 and a batch size of 10. **The Test Error produced was a very low 0.2275. This error could be improved even further the more times the code is run because of the nature of NNs being stochastic and having randomness within its computation (this is why we exported the best model out of many runs).**

2.7.5 Using SGD Optimiser

Despite the fact that when using other optimizers that were not Adam a much higher MSE was produced, it’s important to note that the Test Data used is only a 5% size sample of the actual Test Dataset. This means that optimizers that perform better on larger datasets may potentially lead to smaller errors on the full-sized Test Data. Whilst SGD does not explicitly improve as dataset size increases, we include one test result of SGD just as a precaution for the unknown full dataset.

2.7.6 Best Test Error (MSE)

For our Neural Network Model, a Test Error of 0.2275 was obtained (as shown in Section 2.7.4).

Results

A summary of all of the results calculated within the Methodology is shown in the table below:

Model	**Best Test Error (MSE) Produced**
Linear Regression	0.3089
Lasso Regression	0.9816
Ridge Regression	0.3089
Elastic Net CV	0.9858
KNN Classifiers	0.2502
Neural Network	0.2275

Out of all the models created, a clear winner for lowest Test Error goes to Neural Networks with an MSE of 0.2275.

Conclusion

In conclusion, multiple successful models were designed to predict the ATM cash demand of banks based on the ‘Withdraw’ response variable of the dataset provided. The result of very precise Test Errors from the majority of our models are subject to a variety of techniques used, such as EDA into interaction effects, cross validation and a large amount of model-specific optimization such as testing various neighbours for KNN. The final results showed that out of all of the models used, the most accurate model (by MSE) is the finely-tuned Neural Network model, whilst KNN Classifiers were a close second. Both of these processes are computationally expensive however, and hence Linear Regression could be employed to produce a slightly less accurate but highly efficient model if required.