Predicting a BUY or SELL from a Trade of a Member of the US House of Representatives

Derek Wen and Raymond Song

Summary of Findings

Introduction

We will be predicting whether a trade given the stock trades by members of the US House of Representatives is a `BUY` or `SELL`. Therefore, we are conducting `classification` using `binary classification`. The response variable for our project (what we are predicting) is `whether the trade is a BUY or SELL`. We chose this response variable because it would help us answer the question of if buying and selling occured at the same rate regardless of purchase amount or owner. We are curious if there are certains trends or patterns that influences a trade to be bought or sold. The metric we will be using to evaluate our model will be `accuracy` because all incorrect predictions are weighted the same for our problem. We likely won't know if the capital gains are over $200 USD because usually this only occurs with sales. Therefore in our data cleaning process, we drop `ptr_link` and `cap_gains_over_200_usd` as they serve no purpose. We would know the representative, the type of owner, the ticker, the asset description, the amount purchased or sold, and the transaction dates because they are known as the transaction is made.

Baseline Model

For our Baseline Model, we picked `amount` and `representative` to be our two features. We have one ordinal feature (`amount`) and one nominal feature (`representative`). We performed ordinal encoding onto `amount` and one-hot encoding onto `representative`. For our one-hot encoder, we set the `drop` argument to `first` to remove redundancy and the `handle_unknown` argument to `ignore` to ignore any values that are in the testing data set but not in the training data set. We picked a RandomForestClassifier using `10` for our n_estimators and max_depth.

For our results, the accuracy of our `training` data set was around $61%$ while the accuracy for our `test` data set was around 61% as well. The proportion of `purchase` in our test split turns out to be around 51%. Our model performs better than guessing whether a trade is a buy or sell (a 50% chance), which is why we believe our model is "good". Even in unseen data our accuracy was around 60%, greater than the chance of randomly choosing.

Final Model

For our new features, we decided to create a feature, `transaction_date`, to keep track of the type of day that the stock was traded, a feature, `state`, to obtain the state of the district, and a feature, `Party`, for the political association of the Congresssmen that made the trade. For `transaction_date`, knowing what type of day it was when a trade occured could be beneficial in the fact that there could be a pattern for a certain type of trade for certain days. For instance, many investors may buy more often than sell on Monday because that is when the market opens after a period of closure (Saturday and Sunday is when the market closes) and is when investors obtain their paycheck to spend money with. There may be some days where there is a much higher volume of stocks sold rather than stocks bought, which is important to factor in for our analysis. For `state`, knowing the state in which a Congressman is from could help determine whether a trade is sell or buy since there may be a possibility that Congressmen from certain states tend to buy more rather than sell or vice versa. For instance, congressmen from California may possibly buy more stocks rather than sell stocks due to the amount of resources and wealth they have compared to other states. For `Party`, one's political affiliation with a party may help determine how frequently they buy or sell a stock. One party may tend to purchase more stocks rather than sell or vice versa. For instance, the Democratic Party members may tend to buy more stocks relating to clean energy.

For our model, we picked a `RandomForestClassifier`. We performed a GridSearch to find the best hyperparameters of `max_depth` and `n_estimators`. We find our best parameters for the `RandomForestClassifier` are 40 for `max_depth` and 15 for `n_estimators`. We obtain a `training` accuracy of around 70% and a `test` set accuracy of around 65%.

Fairness Analysis

We perform a fairness analaysis to answer `whether our model performs worse for members of the Democratic party compared to members of the Republican party`. We perform a `permutation test` and use `accuracy` across our two groups.

`Null Hypothesis`: Our model is fair. Its accuracy for members of the Democratic party and members of the Republican party are roughly the sane, and any differences are due to random chance.

`Alternate Hypothesis`: Our model is unfair. Its accuracy for members of the Democratic Party is different than its accuracy for members of the Republican Party.

`Test Statistics`: The absolute difference in accuracy between the two parties. We used difference in accuracy because our falsely predicting sale or falsely predicting purchase does not make much of a differnce. The only thing we want to know if if the accuracy for the different parties was due to random chance or not.

`Significance Level`: 0.05

After running our permutation test, we obtain a p-value of 0 which is less than our significance level and `reject the null hypothesis` that our model is fair in determining the transaction type for Democrats and Republicans.

Code

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import seaborn as sns
%config InlineBackend.figure_format = 'retina'  # Higher resolution figures
```

Cleaning

We read the dataset in from CSV and assign it to the variable `raw_stocks`

In [ ]:
```
raw_stocks = pd.read_csv("all_transactions.csv")
raw_stocks.head()
```

Out[ ]:

	disclosure_year	disclosure_date	transaction_date	owner	ticker	asset_description	type	am
0	2021	10/04/2021	2021-09-27	joint	BP	BP plc	purchase	1,0 — 1!
1	2021	10/04/2021	2021-09-13	joint	XOM	Exxon Mobil Corporation	purchase	1,0 — 1!
2	2021	10/04/2021	2021-09-10	joint	ILPT	Industrial Logistics Properties Trust - Common...	purchase	15, — 50
3	2021	10/04/2021	2021-09-28	joint	PM	Phillip Morris International Inc	purchase	15, — 50
4	2021	10/04/2021	2021-09-17	self	BLK	BlackRock Inc	sale_partial	1,0 — 1!

We found the values of `'--'` to denoate missing values in the dataset. Therefore, we replaced them with `np.NaN` instead.

In [ ]:
```
stocks = raw_stocks.replace("--", np.NaN)
```

In [ ]:
```
stocks['disclosure_date'] = pd.to_datetime(stocks['disclosure_date']) # convert the co
stocks['transaction_date'] = stocks['transaction_date'].str.replace('2?0[0-9]{3}[-]',
stocks['transaction_date'] = pd.to_datetime(stocks['transaction_date']) # convert the
```

We simplified the problem to just categorizing a `purchase` and `sale` instead of `purchase`, `partial sale`, `full sale`, and `exchange`. We convert `sale_full` and `sale_partial` to `sale` and filter out `exchange`.

In [ ]:
```
stocks = stocks.replace({"sale_full":"sale", "sale_partial":"sale"}) # replace sale_fu
```

In [ ]:
```
stocks = stocks[stocks['type'] != 'exchange'] # we filter out exchange
```

The columns `ptr_link` and `cap_gains_over_200_usd` serve no purpose for our needs and are redudant. Therefore they are dropped from the dataset.

In [ ]:
```
stocks = stocks.drop(columns = ["ptr_link", "cap_gains_over_200_usd"])
```

The `amount` column contains overlaps between intervals of the amount purchased in a trade. For instance, there are values labeled as `$1,000,000+` while other values are labeled as `$1,000,001 - $5,000,000` or `$5,000,001 - $25,000,000`. The `$1,000,000+` value can widly vary within each individual value. Therefore we combined all intervals greater than `$1,000,000` into the `$1,000,000+` interval for simplicty sake. In addition, we found minor errors in some of the intervals. One interval was labeled `$1,001 -` when it should have been `$1,001 - $15,000`. Because of this minor error, we corrected this mistake and replaced the rest of the intervals by adding an additional dollar (`$1,000 - $15,000` went to `1,001 - $15,000` and `15,000 - $50,000` went to `15,001 - $50,000`).

In [ ]:
```
stocks = (stocks.replace({"$1,001 -":"$1,001 - $15,000",
        "$1,000 - $15,000":"$1,001 - $15,000",
        "$15,000 - $50,000": "$15,001 - $50,000",
        "$1,000,001 - $5,000,000": "$1,000,000 +",
        "$5,000,001 - $25,000,000":"$1,000,000 +",
        "$50,000,000 +": "$1,000,000 +",
        "$1,000,000 - $5,000,000": "$1,000,000 +"}))
```

We read a csv file containing the state abbreviations for each state and set it to `state_abbrev`. We also set `state_abbrev_dict` to a dictionary containing the state abbreviation and their full name.

In [ ]:
```
state_abrev = pd.read_csv("state_abrev.csv")
state_index = state_abrev[['State', 'Abrev']].set_index('State')
state_abrev_dict = state_index.to_dict()['Abrev'] # dictionary containing the state ab
```

We read in a csv file containing the party affiliation for each person in a district and set it to `parties`.

In [ ]:
```
parties = pd.read_csv("party_affil.csv")
dist_party = parties[['District', 'Party']] # dataframe of the district and party affi
clean_state = dist_party.loc[:,'District'].str.replace('at-large', '00') # replacing a
state_name = clean_state.str.extract(r'([\w ]+).\d{1,2}') # obtaining a dataframe with
district_state = state_name.loc[:,0].apply(lambda x: state_abrev_dict[x]) # series wit
district_num = clean_state.str.extract(r'[\w ]+.(\d{1,2})').loc[:,0].apply(lambda x: s
```

We create a district abbreviations series that matches the same patterns of `parties`.

In [ ]:
```
dist_abrev = district_state + district_num
dist_abrev.name = 'dist_abrev'
```

We are able to merge the districts with the affiliated parties to create `parties_merge`.

In [ ]:
```
party_merge = pd.concat([dist_party, dist_abrev], axis=1).drop('District', axis=1)
party_merge.head()
```

Out[ ]:

	Party	dist_abrev
0	Republican	AL01
1	Republican	AL02
2	Republican	AL03
3	Republican	AL04
4	Republican	AL05

We merge `party_merge` and `stocks` to merge in the affiliated party with each representative and district. We are left with a cleaned stocks dataset shown below with the designated party.

In [ ]:
```
stock_party = stocks.merge(party_merge, left_on='district', right_on='dist_abrev', hov
stock_party = stock_party.drop('dist_abrev', axis=1)
stock_party.head()
```

Out[ ]:

	disclosure_year	disclosure_date	transaction_date	owner	ticker	asset_description	type	amou
0	2021	2021-10-04	2021-09-27	joint	BP	BP plc	purchase	1,001 – 15,0
1	2021	2021-10-04	2021-09-13	joint	XOM	Exxon Mobil Corporation	purchase	1,001 – 15,0
2	2021	2021-10-04	2021-09-10	joint	ILPT	Industrial Logistics Properties Trust - Common...	purchase	15,00 – 50,0
3	2021	2021-10-04	2021-09-28	joint	PM	Phillip Morris International Inc	purchase	15,00 – 50,0
4	2021	2021-11-02	2021-10-13	joint	MO	Altria Group Inc	purchase	1,001 – 15,0

Baseline Model

For our Baseline Model, we picked `amount` and `representative` to be our two features. We will use an ordinal encoder for `amount` and an one hot encoder for `representative`. We picked a RandomForestClassifier using 10 for our n_estimators and max_depth.

In [ ]:
```
#Importing python packages
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

X = stock_party[["amount", "representative"]] # defining X to be a dataset with our tw
y = stock_party['type'] # defining Y to be what we are predicting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20) # splting d
col_tran = ColumnTransformer([
    ('ordinal', OrdinalEncoder(), ['amount']),
    ('one-hot', OneHotEncoder(drop="first", handle_unknown="ignore"), ['representative
]) # setting up a column transformer where we use ordinal encoder for amount and onehc
pl = Pipeline(
    [("col_tran", col_tran),
    ("classifier",  RandomForestClassifier(n_estimators=10, max_depth=10))
    ]
) # creating a pipline with our column transformer and randomforestclassifier
pl.fit(X_train, y_train) # fitting our data
```

Out[ ]:
```
Pipeline(steps=[('col_tran',
                 ColumnTransformer(transformers=[('ordinal', OrdinalEncoder(),
                                                  ['amount']),
                                                 ('one-hot',
                                                  OneHotEncoder(drop='first',
                                                                handle_unknown='ignor
e'),
                                                  ['representative'])])),
                ('classifier',
                 RandomForestClassifier(max_depth=10, n_estimators=10))])
```

For our results, the accuracy of our `training` data set is around 61%.

In [ ]:
```
(pl.predict(X_train) == y_train).mean() # accuracy of our training data set
```

Out[ ]:
```
0.5982396870554765
```

The accuracy for our `test` data set is around 61% as well.

In [ ]:
```
import warnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    print((pl.predict(X_test) == y_test).mean()) # accuracy of our test data set
```

```
0.594950213371266
```

The proportion of `purchase` in our test split turns out to be around 51.5%.

In [ ]:
```
(y_test == 'purchase').mean() # Proportion of `purchase` in our test split
```

Out[ ]:
```
0.5295163584637269
```

Final Model

We first create helper functions to transform our columns into new features. `get_dow` is a function that takes in a dataframe and transforms the `transaction_date` column into values for the day of the week, returning a new dataframe in the process. `get_state` is a function that takes in a dataframe and transforms the `state` column to obtain the first two letters of the string, obtaining the state of a district and returning the same datafrane back.

In [ ]:
```
def get_dow(df):
    new_df = pd.DataFrame(df.loc[:,"transaction_date"].apply(lambda x: x.dayofweek))
    return new_df
```

In [ ]:
```
def get_state(df):
    df['state'] = df.loc[:, 'district'].str.slice(start=0, stop=2)
    return df
```

For our Final Model, we picked `amount`, `representative`, `transaction_date`, `state`, and `Party` to be our features. We will use an ordinal encoder for `amount`, a function transformer for `transaction_date`, and an one hot encoder for `representative`, `state`, and `Party`. We picked a `RandomForestClassifier` using 10 for our n_estimators and max_depth as placeholders before we use GridSearch and finding the best hyperparameters that ended up performing the best.

In [ ]:
```
#Importing python packages
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import FunctionTransformer

X = stock_party[["amount", "representative", "transaction_date", "district", "Party"]]
y = stock_party['type'] # defining Y to be what we are predicting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state
col_tran = ColumnTransformer([
    ('dayofweek', FunctionTransformer(get_dow), ['transaction_date']),
    ('ordinal', OrdinalEncoder(), ['amount']),
    ('one-hot', OneHotEncoder(drop="first", handle_unknown="ignore"), ['representative
]) # setting up a column transformer where we use ordinal encoder for amount and oneho
pl = Pipeline(
    [
        ("get_state", FunctionTransformer(get_state)),
        ("col_tran", col_tran),
        ("classifier",  RandomForestClassifier(n_estimators=10, max_depth=10))
    ]
)
```

We create our set of `hyperparameters` for our `n_estimators` and `max_depth`.

In [ ]:
```
hyperparameters = ({'classifier__n_estimators': [1, 2, 3, 4, 5, 10, 15, 25, 50],
                    'classifier__max_depth': [2, 4, 6, 8, 10, 20, 40, 50, 100]})
```

We perform a GridSearch with `hyperparameters` and our pipeline to obtain the hyperparameters that ended up performing the best.

In [ ]:
```
from sklearn.model_selection import GridSearchCV
import warnings
```

```python
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    search = GridSearchCV(pl, hyperparameters, cv=5)
    search.fit(X_train, y_train)
```

We find our best parameters for the `RandomForestClassifier` are 40 for `max_depth` and 15 for `n_estimators`

In [ ]:
```python
search.best_params_
```

Out[ ]:
```
{'classifier__max_depth': 40, 'classifier__n_estimators': 15}
```

In [ ]:
```python
best_pl = search.best_estimator_
```

We get around 70% accuracy on the `training` split.

In [ ]:
```python
(best_pl.predict(X_train) == y_train).mean() # accuracy of our training data set
```

Out[ ]:
```
0.7083926031294452
```

We get around 65% accuracy on the `test` split.

In [ ]:
```python
import warnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    print((best_pl.predict(X_test) == y_test).mean()) # accuracy of our test data set
```

```
0.6578947368421053
```

Fairness Analysis

We perform a fairness analaysis to answer `whether our model performs worse for members of the Democratic party compared to members of the Republican party`. We perform a `permutation test` and use `accuracy` across our two groups.

`Null Hypothesis`: Our model is fair. Its accuracy for members of the Democratic party and members of the Republican party are roughly the sane, and any differences are due to random chance.

`Alternate Hypothesis`: Our model is unfair. Its accuracy for members of the Democratic Party is lower than its accuracy for members of the Republican Party.

`Test Statistics`: The absolute difference in accuracy between the two parties. We used difference in accuracy because our falsely predicting sale or falsely predicting purchase does not make much of a differnce. The only thing we want to know if if the accuracy for the different parties was due to random chance or not.

`Significance Level`: 0.05

We use our entire dataset with our best model because we want to know if it predicts better for one party than another. We set `perm_data` to be a copy of our data set and `y_pred` for our predicted values using our best model

In [ ]:
```
perm_data = X.copy() # create a copy of our data
```

In [ ]:
```
import warnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    y_pred = best_pl.predict(X) # obtaining the predicted values from our pipeline
```

We first caclulate the accuracy for our current data

In [ ]:
```
import warnings
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    perm_data['isCorrect'] = best_pl.predict(X) == y # obtain the accuracy for our cur
```

We set the absolute difference in accuracy between Republican and Democrat for our data set to be `abs_diff_data`

In [ ]:
```
abs_diff_data = np.abs(perm_data.groupby('Party').mean().diff().iloc[1, 0]) # Find abs
abs_diff_data
```

Out[ ]:
```
0.09039730079334041
```

We run our permutation test 10,000 times and set our simulated values for the absolute difference to be `perm_accs`

In [ ]:
```
num_perms = 10000
perm_diffs = []
for _ in range(num_perms):
    perm_data['Party'] = perm_data['Party'].sample(frac=1).reset_index(drop=True)
    abs_diff_perm = np.abs(perm_data.groupby('Party').mean().diff().iloc[1, 0])
    perm_diffs.append(abs_diff_perm)
perm_accs = pd.Series(perm_diffs)
```

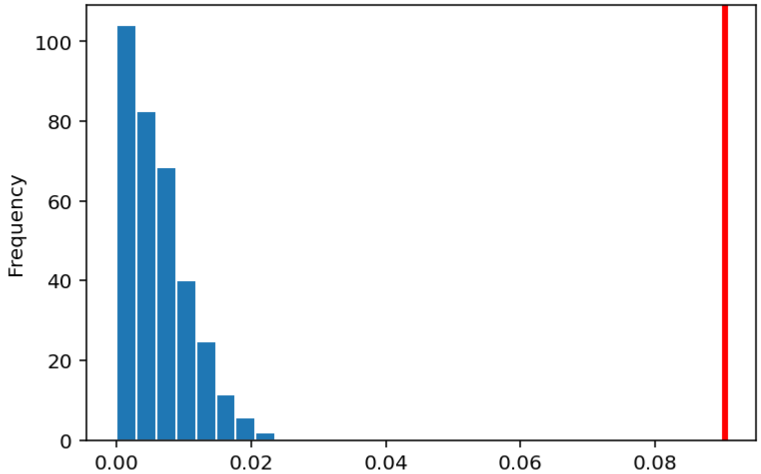
We obtain a `p-value` of 0.0 shown below

In [ ]:
```
(perm_accs >= abs_diff_data).mean() # P-value calculation
```

Out[ ]:
```
0.0
```

Finally, we graph a histogram to show how our difference of accuracy compares to simulated values.

In [ ]:
```
perm_accs.plot(kind="hist", density=True, ec='w', bins=10)
plt.axvline(x=abs_diff_data, color='red', linewidth=3)
plt.show()
```



After running our permutation test, we obtain a p-value less than our significance level and `reject the null hypothesis` that that our model is fair.