Sussy stock trades

Prabina Pokharel, Atharva Kulkarni

Summary of Findings

Introduction

We've decided to create a binary classification model for this project. Specifically, we are going to try to predict whether a given stock transaction resulted in capital gains over $200 or not.

Response variable: `cap_gains_over_200_usd`

Validation parameter: Precision

Baseline Model

We used 2 nominal variables to predict the response variable, `state` and `type`. We did not need to perform any cleaning/feature engineering in this model, because `state` and `type` were given to us. We achieved a precision scores in between 0.75 - 0.85 on *unseen* data, so we think that even this baseline model does a decent job at predicting values as true only if they are true in real life most of the time.

Final Model

We used 5 variables to predict the response variable:

1. `state` (from baseline), nominal
2. `type` (from baseline), nominal
3. `owner`, nominal
4. `amount_cleaned` (a feature we created), numerical
    - we added this feature because the more money you have invested, it is more likely a sell will result in capital gains of over $200.
5. `non_disclosure_period(days)` (a feature we created), numerical
    - we added this feature because if the `non_disclosure_period(days)` is high, we believe that corresponds to a trade that a representative does not want to disclose because they have higher capital gains.

We chose Random Forest Classifier for our model. Since our model was a classification problem, we could either choose Decision Tree or Random Forest. We decided to go with Random Forest since it is made out of many Decision Trees, model performance should be higher.

After performing GridSearchCV on our unfit Random Forest Classifier, these are the hyperparamters we found to be optimizing precision score:

- criterion': 'entropy',
- max_depth': 5,
- min_samples_split': 5,
- n_estimators': 30

On our final model, our precision score increases from 0.75-0.85 to 0.9+, so we definitely believe that the 3 additional features we chose are very correlated with our response variable.

Fairness Analysis

We ran permutation test on the result we got from our model to check if there is any parity towards a specific group. More specifically, we tried to answer whether the model is unfair towards Californian reps when compared to non-Californian reps. Our null hypothesis is that the model is fair and our alternate hypothesis is that the model is unfair towards Californian reps. We used precision as a measure of parity. If the measure yield same number for both groups, then our model likely achieves precision parity. Our observed precision difference was around 95%. To determine if 95% difference is significant, we ran a permutation test and got a p-value of less than our chosen cutoff of 0.01, we showed that our model is likely unfair towards Californian reps.

Code

In [15]:

```
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import seaborn as sns
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import FunctionTransformer, OneHotEncoder, Standar
from sklearn.compose import ColumnTransformer
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import recall_score, precision_score
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings('ignore')
%config InlineBackend.figure_format = 'retina'  # Higher resolution figures
```

Framing the Problem

Before we get to framing our problem, here's an overview of our cleaning steps and feature engineering from project 3:

- Changing `disclosure_date` and `transaction_date` to pandas datetime object
  - This was handy to create, as we could simply subtract the two columns to create a new new `non_disclosure_period(days)` column.
- Removed the 'Hon.' out of representatives' names
  - To make it easier to read.
- Added a `state` column, created from `district`
  - Necessary in order to be able to answer our main question!
  - Also useful to create cool aggregates by state!
- Added an `amount_cleaned` column, created from `amount`
  - This was important in order to be able to do any sort of math calculations with relation to amount of the transaction value.

We've decided to create a binary classification model for this project. Specifically, we are going to try to predict whether a given stock transaction resulted in capital gains over $200 or not.

Response variable: `cap_gains_over_200_usd`

Figuring our what metric to use is a bit trickier. First we'd need to see if our response variable data distribution is imbalanced or not:

In [16]:
```
df = pd.read_csv('data/cleaned_transactions.csv')
df['cap_gains_over_200_usd'].value_counts()
```

Out[16]:
```
False    13238
True       964
Name: cap_gains_over_200_usd, dtype: int64
```

So unfortunately we don't have 50/50 balanced data. When it comes to unbalanced data, we can choose between recall or precision. Since we care about false positives in our prediction (i.e., we don't want to just classify every trade as having a positive response variable value and having many false positives), we will choose **precision** as our validation parameter.

Here's why we're *not* choosing...

- Accuracy: since our `False` values in our response variable data accounts for nearly 93% of the total response variable data, we could achieve 93% accuracy by just predicting all values are `False` .
- Recall: seeimingly similar to precision, recall only takes into account true positives and false negatives. We could have also chosen recall as our validation parameter, but decided not to because we are placing more importance on predicting values as positive if they really are positive.

We won't have any 'time of prediction' issues because all of the columns in our dataset, and any extra features we engineer, because all of them can be found out before finding out `cap_gains_over_200_usd` .

Baseline Model

For our baseline model, we're using 2 variables to predict the response variable, `state` and `type`. Both are categorical.

- state: we believe state actually does matter in predicting capital gains over 200 because in our project 3 permutation test, we determined representatives from some states are more likely to have `cap_gains_over_200_usd` as `True` than other states.
- type: if a transaction is not some sort of sale, then it is impossible for `cap_gains_over_200_usd` to be `True`.

Let's split up the data into X and y, and then conduct a train-test-split.

In [17]:
```
X = df[['state', 'type']]
y = df['cap_gains_over_200_usd']
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2)
```

Our `base_preproc` is a column transformer that's going to OHE the `state` and `type` columns.

In [18]:
```
base_preproc = ColumnTransformer(
    transformers = [
        ('ohe', OneHotEncoder(), ['state', 'type'])
    ],
    remainder='drop'
)
```

We then put the column transformer into a pipeline, paired with a `RandomForestClassifier` object. Note that the `RandomForestClassifier` has no parameters passed in, so it will use default values. We will conduct GridSearchCV for the best combination of hyperparameters in the next section.

In [19]:
```
base_pl = Pipeline([
    ('prepocessor', base_preproc),
    ('dec-tree', RandomForestClassifier())
])

# fit the pipeline
base_pl.fit(X_train, y_train);
```

In [20]:
```
# predict response variables values based on fitted pipieline
y_pred_base = base_pl.predict(X_test)

print(f'Testing Precision: {metrics.precision_score(y_test, y_pred_base):.2f}')
```

```
Testing Precision: 0.83
```

Final Model

For our final model, we're using 5 variables to predict the response variable:

1. `state` (from baseline), nominal
2. `type` (from baseline), nominal
3. `owner` , nominal
4. `amount_cleaned` (a feature we created), numerical
5. `non_disclosure_period(days)` (a feature we created), numerical

How we feature engineered the 2 new columns:

- `amount_cleaned`
  - We created a dictionary to map range values to their average value, and then replaced range values in `amount` with its average amount.
- `non_disclosure_period(days)`
  - We changed `disclosure_date` and `transaction_date` to pandas datetime object. This was handy to create, as we could simply subtract the two columns to create the new feature.

Now let's deal with missing values in the 5 variables we're using for prediction, and make sure all column types are the types that we want.

In [21]:
```
# '--' does not signify anything so replace with nan
df['owner'] = df['owner'].replace({'--': np.nan})

# np.nan is not acceptable in sklearn pipelines so we replace it with its own
# categorical value
df = df.replace({np.NaN:'missing'})

# since we imported from csv, make sure 'disclosure_date' and 'transaction_da
# are of type date time
df['disclosure_date'] = pd.to_datetime(df['disclosure_date'], errors = 'coerc
df['transaction_date'] = pd.to_datetime(df['transaction_date'], errors = 'coe

# we are only going to include rows in which the 'non_disclosure_period(days)
# values are not missing
df = df[df['non_disclosure_period(days)']!='missing']
df.head()
```

Out[21]:

	disclosure_year	disclosure_date	transaction_date	owner	ticker	asset_description	
0	2021	2021-10-04	2021-09-27	joint	BP	BP plc	purc
1	2021	2021-10-04	2021-09-13	joint	XOM	Exxon Mobil Corporation	purc
2	2021	2021-10-04	2021-09-10	joint	ILPT	Industrial Logistics Properties Trust - Common...	purc

	disclosure_year	disclosure_date	transaction_date	owner	ticker	asset_description	
3	2021	2021-10-04	2021-09-28	joint	PM	Phillip Morris International Inc	purc

Let's split up the data into X and y, and then conduct a train-test-split.

In [22]:
```
X = df[['owner', 'type', 'state', 'amount_cleaned', 'non_disclosure_period(da
y = df['cap_gains_over_200_usd']
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2)
```

Let's create a ColumnTransformer object with OneHotEncoder and StandardScalar as its transformers.

In [23]:
```
final_preproc = ColumnTransformer(
    transformers = [
        ('ohe', OneHotEncoder(drop='first'), ['owner', 'type', 'state']),
        ('std', StandardScaler(), ['amount_cleaned', 'non_disclosure_period(d
    ],
    remainder='drop'
)
```

In [24]:
```
final_pl = Pipeline([
    ('prepocessor', final_preproc),
    # these hyperparameter values are from conducting GridSearchCV (see below
    ('dec-tree', RandomForestClassifier(criterion='entropy', max_depth=5,min_
])
final_pl.fit(X_train, y_train);
```

In [25]:
```
y_pred_final = final_pl.predict(X_test)
print(f'Testing Precision: {metrics.precision_score(y_test, y_pred_final):.2f
```

```
Testing Precision: 0.94
```

Let's conduct GridSearchCV to see which combination of hyperparamters is best. This section appears after instantiating and fitting the pipeline, but we conducted GridSearchCV before fitting final_pl.

In [26]:
```
hyperparameters = {
    'dec-tree__n_estimators': np.arange(10, 100, 20),
    'dec-tree__criterion': ['gini', 'entropy'],
    'dec-tree__max_depth': [2,3,4,5],
    'dec-tree__min_samples_split': [1,2,5]
}
```

In [27]:
```
searcher = GridSearchCV(final_pl, param_grid = hyperparameters, cv=5, scoring
searcher.fit(X_train, y_train)
searcher.best_params_
```

Out[27]:
```
{'dec-tree__criterion': 'entropy',
 'dec-tree__max_depth': 5,
```

```
 'dec-tree__min_samples_split': 5,
 'dec-tree  n estimators': 90}
```

Fairness Analysis

We're going to split our dataset into representatives from California and representatives not from California.

Null Hypothesis: Our model is fair. Its precision for Californian representatives and non-Californian representatives are roughly the same, and any discrepancies we see are due to random chance.

Alternative Hypothesis: Our model is unfair. Its precision for Californian representatives is lower than its precision for non-Californian representatives

We will choose a p-value of 0.01. Here, we could have chosen a p-value of 0.05 but we wanted to be extra sure that our model is fair or unfair.

In [37]:
```
results = X_test # we will only focus on testing data
results['prediction'] = y_pred_final # we are setting prediction column to be
results['cap_gains_over_200_usd'] = y_test
results['is_CA'] = (results.state == 'CA').replace({True: 'cali', False: 'non
results.head(3)
```

Out[37]:

	owner	type	state	amount_cleaned	non_disclosure_period(days)	prediction	cap_
4575	joint	purchase	VA	8000.5	8.0	False	
10164	joint	purchase	NC	8000.5	108.0	False	
12437	self	purchase	CA	32500.5	31.0	False	

Here, we observe that the difference in precision between Californian and non-Californian reps is roughly 95%.

In [30]:
```
obs = results.groupby('is_CA').apply(lambda x: metrics.precision_score(x['cap
obs
```

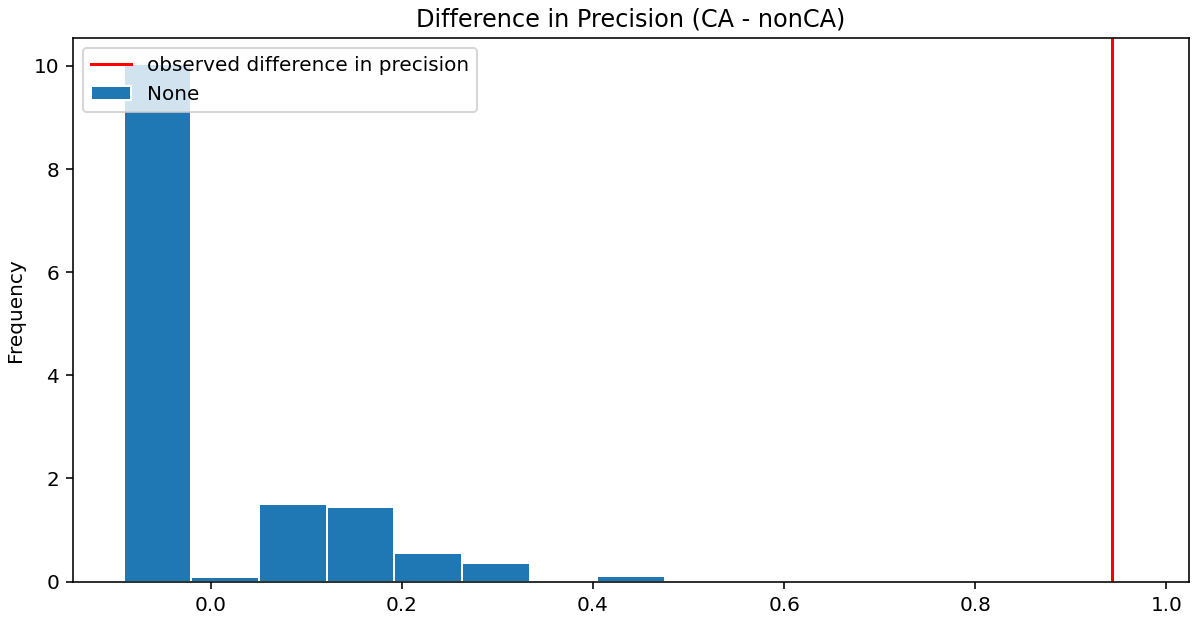
Out[30]:
```
0.9428571428571428
```

We will run a permutation test of 1000 iterations to see if the difference in precision is significant.

In [31]:
```
diff_in_prec = []
for _ in range(1000):
    s = (
        results[['is_CA', 'prediction', 'cap_gains_over_200_usd']]
        .assign(is_CA=results.is_CA.sample(frac=1.0, replace=False).reset_ind
        .groupby('is_CA')
        .apply(lambda x: metrics.precision_score(x['cap_gains_over_200_usd'],
        .diff()
        .iloc[-1]
    )

    diff_in_prec.append(s) # appens precision difference for each iteration
```

In [32]:
```
plt.figure(figsize=(10, 5))
pd.Series(diff_in_prec).plot(kind='hist', ec='w', density=True, bins=15, titl
plt.axvline(x=obs, color='red', label='observed difference in precision')
plt.legend(loc='upper left');
```



In [36]:
```
p_val = (obs <= diff_in_prec).mean()
p_val
```

Out[36]: 0.001

From the graph as well as the p-val, it looks like the difference in precision across the two groups is highly significant. Therefore, our model likely does not achieve precision parity.