Predicting Email Openings using Machine Learning

Our goal is to use features from the accounts and white label customer tables (such as EmailTemplateID, EmailStrategyID, and SubjectLine) to provide meaningful insight into our click and open rates. We will use a variety of techniques, such as regresssion, neural networks, clustering, and principal component analysis to model and predict these two metrics as part of a larger effort to construct a fully automated machine learning email strategy pipeline.
- Lars Gartenberg

We begin by importing our data

Displayed below are the first five of the quarter million entires we consider from our Email Accounts table:

In [44]:

```
df = pd.read_csv(r"C:\Users\LG735620\Desktop\Data Science\Datasets\aggregateEmailDataExpansion2.csv")
df[:5]
```

Out[44]:

	acctid	clientid	emailstrategyid	emailtemplateid	mastertemplate	name	subjectline	bannertext	AccountTyp
0	48427094	AMXA74J4OS	O-E50H	I-REM3	WLP2	3rd Reminder Email	Check out the payment options we have for you!	NaN	Digital Onl
1	48427189	AMXA74J5OS	O-E50L	I-REM3	WLP2	3rd Reminder Email	Check out the payment options we have for you!	NaN	Digital Call
2	48427197	AMXA74J5OS	O-E50L	I-REM3	WLP2	3rd Reminder Email	Check out the payment options we have for you!	NaN	Digital Call
3	48368808	AMX74JN0S	O-E38	H-REM2O	Proactive - Oasis Terms and Conditions	Oasis Settlement 2	Let's review your payment options	NaN	Digital Onl
4	48369342	AMXA74J4IS	O-E50J	I-REM2	WLP2	2nd Reminder Email	Important: Your exclusive settlement offer exp...	NaN	Digital Onl

5 rows × 31 columns

Displayed below are the first 5 of a quarter million entires we consider from our WhiteLabel Customer table:

In [45]:

```
wlAccounts = pd.read_csv(r"C:\Users\LG735620\Desktop\Data Science\Datasets\WhiteLabel Accounts.csv",sep='\t')
```

```
wlAccounts[:5]
```

Out[45]:

	acctid	OasisEligible	CareEligible	ReInEligible	Decile	PreCharge
0	45775451	YES	YES	NO	NaN	Y
1	45775452	YES	NO	NO	NaN	N
2	45775453	NO	NO	NO	NaN	Y
3	45775454	NO	NO	NO	NaN	N
4	45775455	YES	NO	NO	NaN	Y

Preliminary Data Exploration

Before building a predictive model, we must build up our intution for the data by looking at its distributions and averages.

In [55]:

```
result.describe()
```

Out[55]:

	acctid	AccountType	Opened	Clicked	SuccessfulLogin	ContainsOffer	ContainsOfferTable	Cont
count	2.258550e+05	225855.000000	225855.000000	225855.000000	225855.000000	225855.000000	225855.000000	
mean	4.846832e+07	0.494671	0.228306	0.026039	0.258790	0.799287	0.788245	
std	3.362904e+05	0.499973	0.419742	0.159251	0.437971	0.400535	0.408553	
min	4.577546e+07	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	4.836969e+07	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	
50%	4.855663e+07	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	
75%	4.863986e+07	1.000000	0.000000	0.000000	1.000000	1.000000	1.000000	
max	4.891610e+07	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	

8 rows × 38 columns

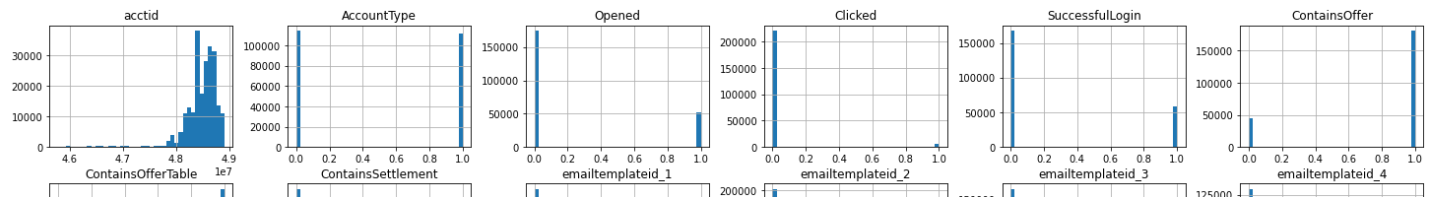
Here, we observe that our email open rate sits at ~23% and that our email click rate is ~2.6%.
Since the click rate is an order of magnitude lower, I had to take a different approach in building out the predicion architecture.
I will write a follow up report detailing this second approach.

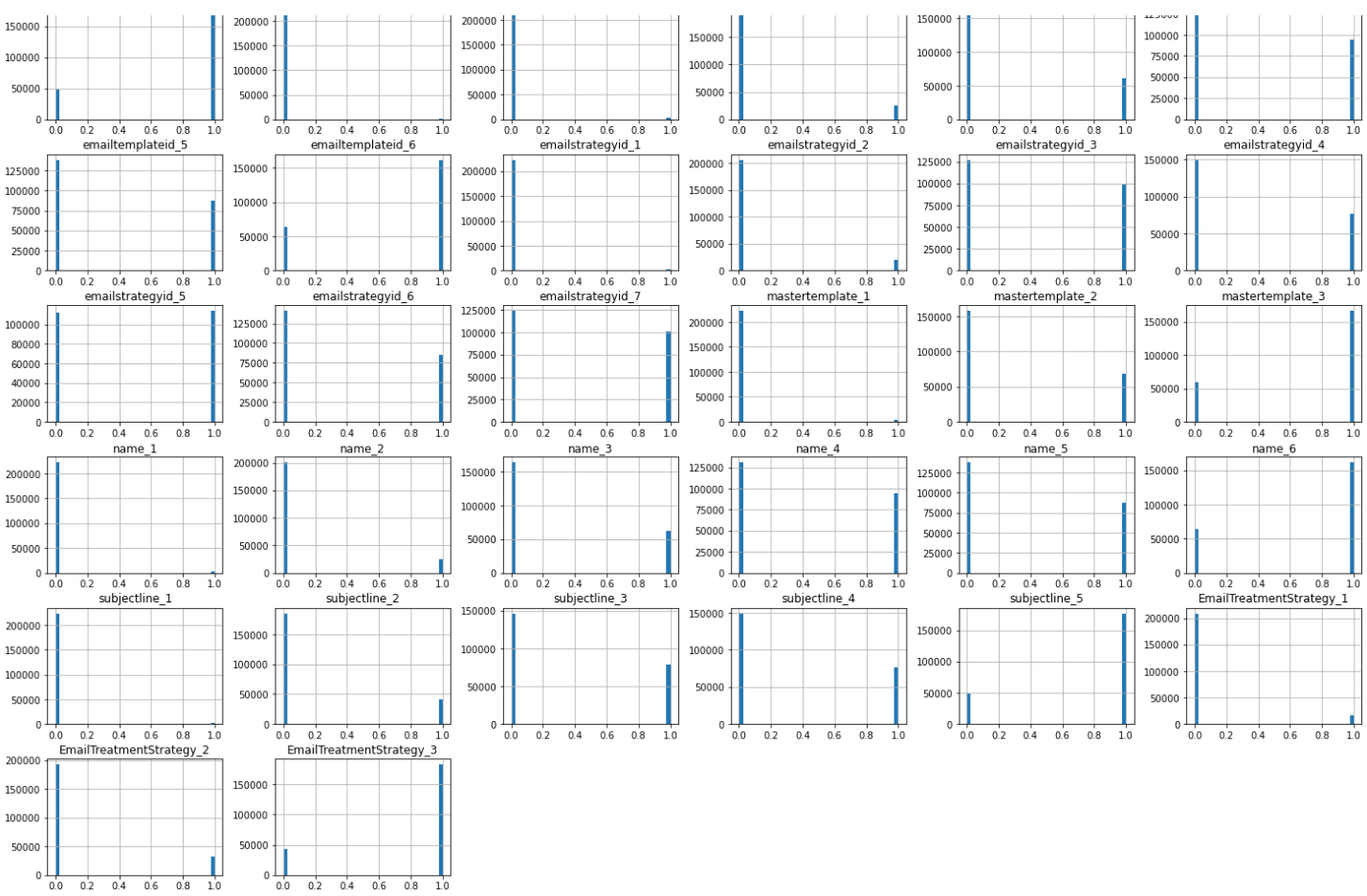
Feature Histograms

Each feature's dsitribtuion can be can be visually represented by their respective hisogram, with up to 70 bins if necesarry.

In [57]:

```
result.hist(bins=40, figsize=(25,20))
plt.show()
```



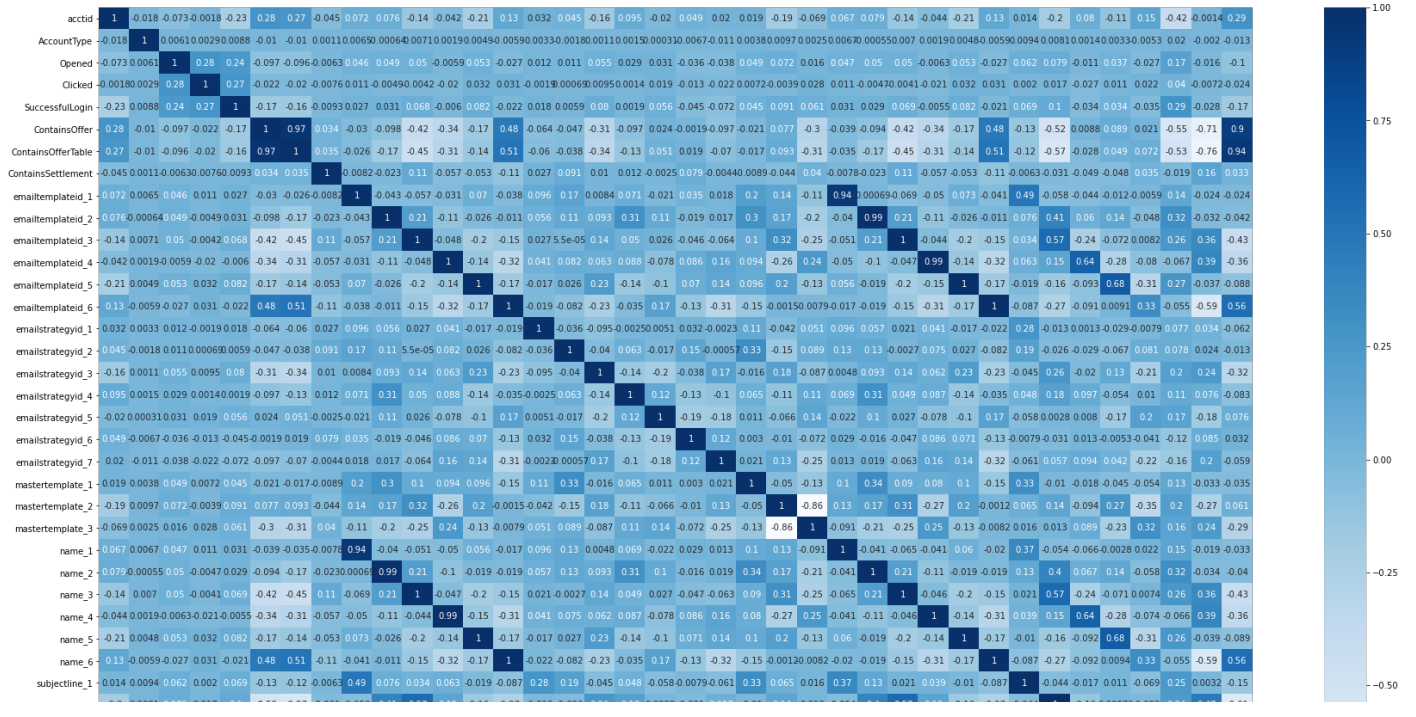


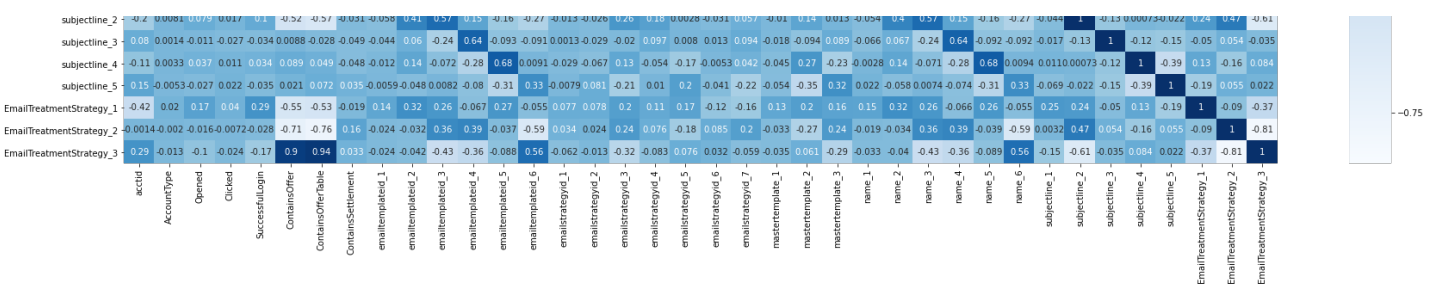
Correlation Matrix

A correlation matrix is a grid that pits each feature against every other feature, and calculates a coefficient in the unit domain [-1,1]. If the coefficient is 1, the two features perfectly map onto each other (i.e. the diagonal that compares each feature to itself). If the coefficient is 0, the two features in question have no correlation. If the feature is close to -1, there exists a strong negative correlation. Depending on the sterngth and magnitude of the coefficients with respect to 'Opened' and 'Clicked', we can observe which features have the strongest linear relationships to these two metrics.

In [54]:

```
plt.figure(figsize=(30,18))
corr = result.corr()
heatmap = sns.heatmap(corr, annot=True, cmap="Blues")
```



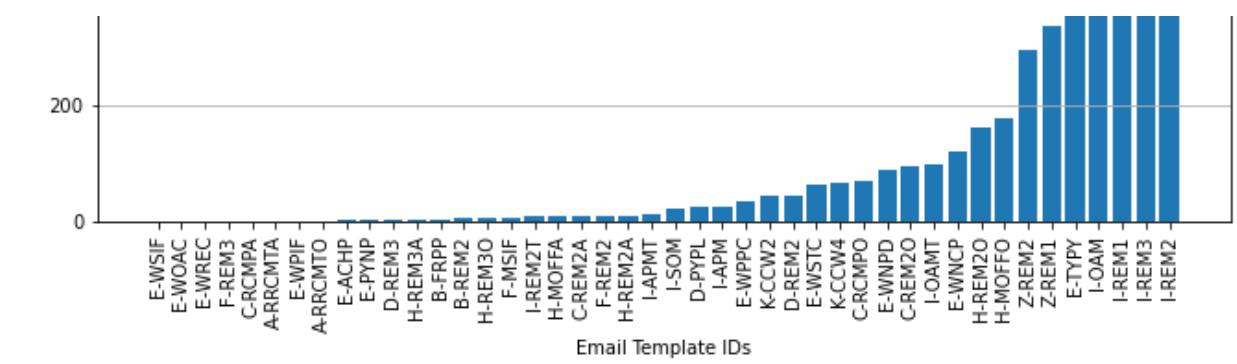


Template ID, Strategies, and Subject Line Distributions

Displayed below are the distributions for Template ID, Strategies, and Subject Line: the three strongest correlations with our two metrics:

In [63]:

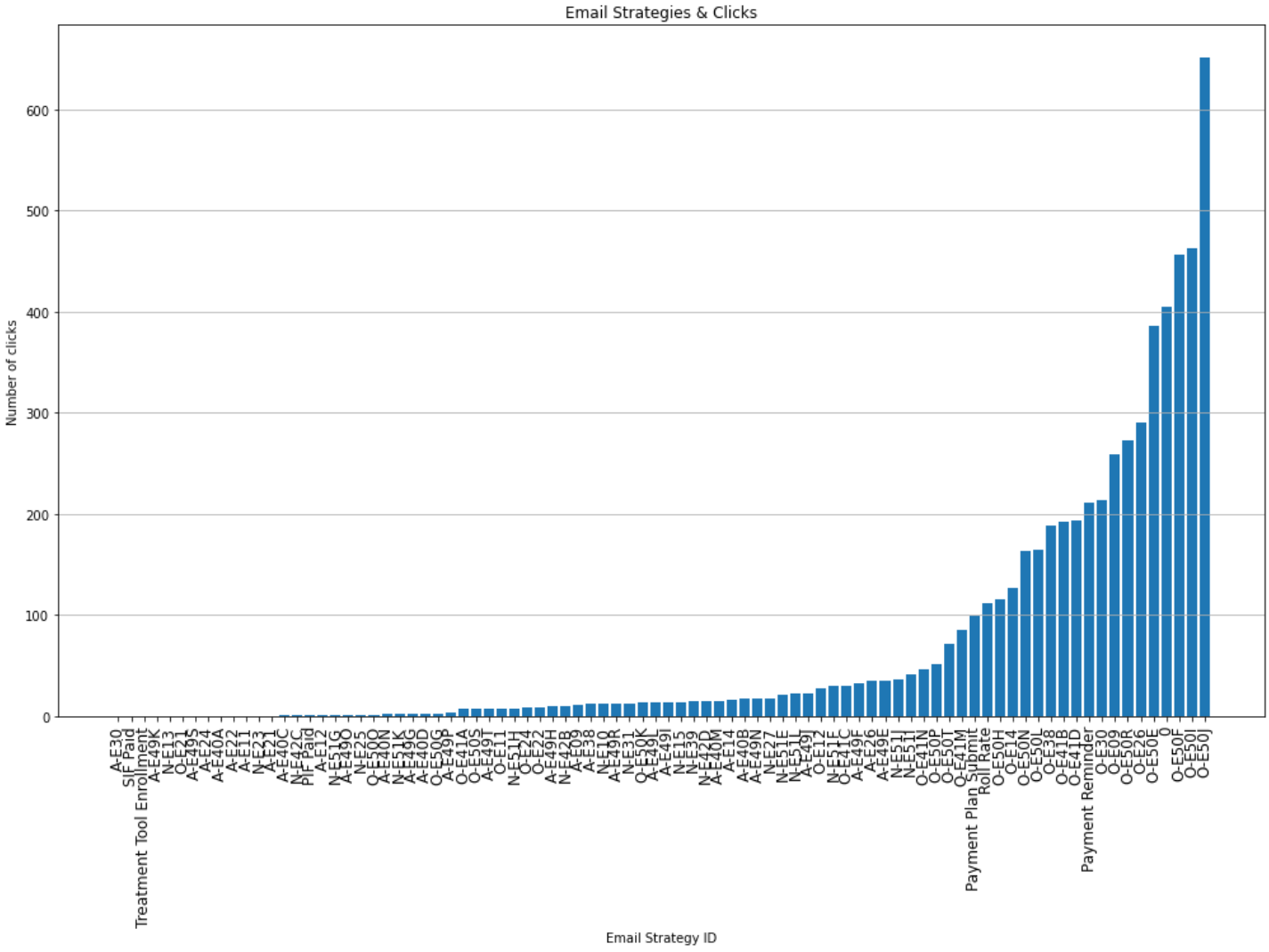
```
data = [df['emailstrategyid'], df['emailtemplateid'],  df['Clicked'], df['subjectline']]
ndf = pd.DataFrame(data=[df['emailstrategyid'], df['emailtemplateid'],  df['Clicked'], d
f['subjectline']])
ndf = ndf.T
ls = ndf['emailstrategyid'].unique().tolist()
stratD = dict.fromkeys(ls, 0)
ls = ndf['emailtemplateid'].unique().tolist()
templateD = dict.fromkeys(ls, 0)
ls = ndf['subjectline'].unique().tolist()
subjectD = dict.fromkeys(ls, 0)

for index,row in ndf.iterrows():
    template = row['emailtemplateid']
    strat = row['emailstrategyid']
    subject = row['subjectline']
    click = row['Clicked']

    templateD[template] += int(click)
    stratD[strat] += int(click)
    subjectD[subject] += int(click)

D = templateD
D = dict(sorted(D.items(), key=lambda item: item[1]))
templateD = D
plt.figure(figsize=(11,8))
plt.bar(range(len(D)), list(D.values()), align='center')
plt.xticks(range(len(D)), list(D.keys()),fontsize = 10,rotation='vertical')
plt.xlabel("Email Template IDs")
plt.ylabel("Number of clicks")
axes = plt.gca()
axes.yaxis.grid()
plt.show()
```



We observe that clicked Email Template IDs follow a gaussian distribution, and that I-type template IDs are overwhelmingly the ones most clicked.

In [61]:

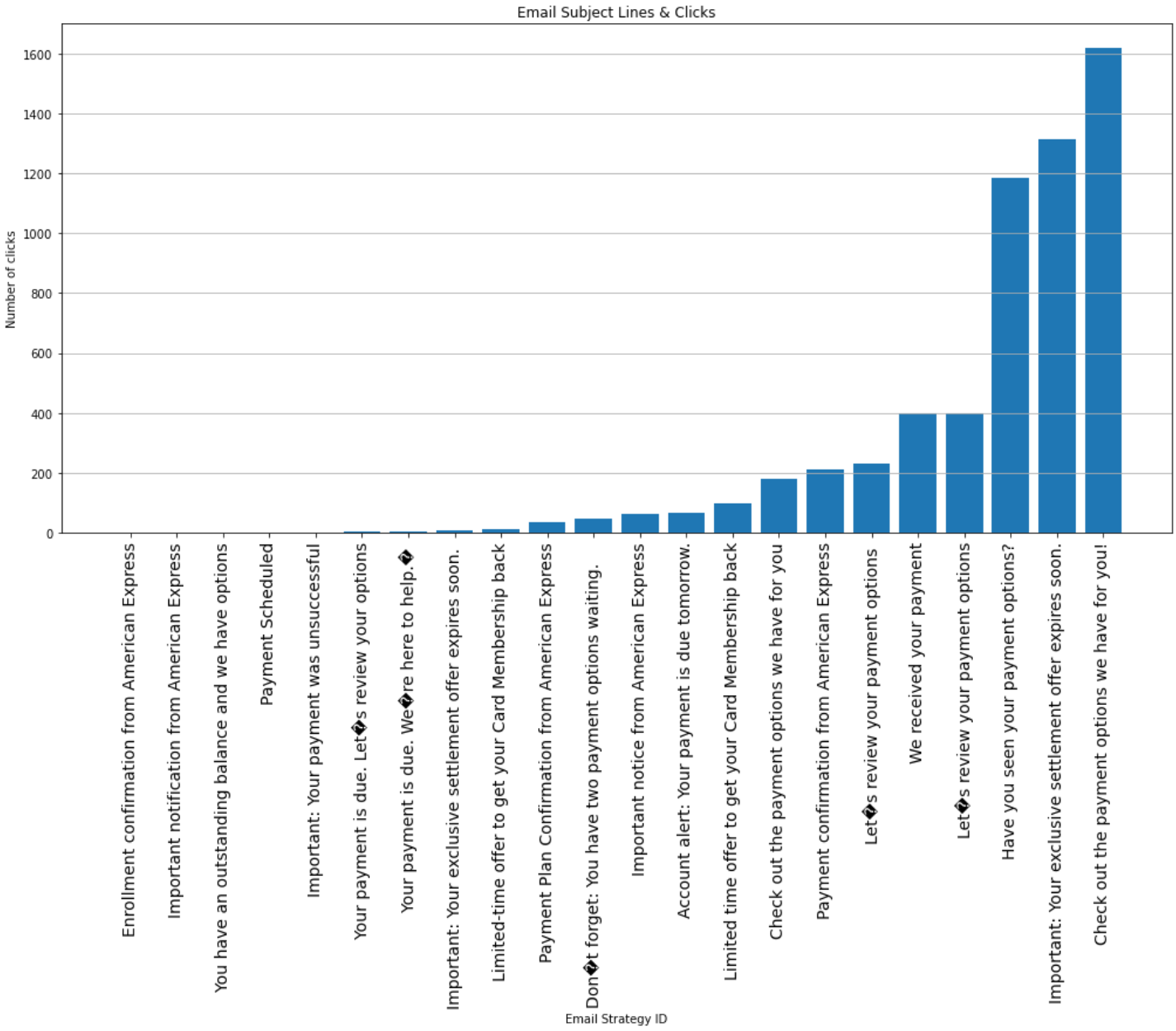
```
D = stratD
D = dict(sorted(D.items(), key=lambda item: item[1]))
plt.figure(figsize=(17,10))
plt.bar(range(len(D)), list(D.values()), align='center')
plt.xticks(range(len(D)), list(D.keys()),fontsize = 12,rotation='vertical')
plt.title("Email Strategies & Clicks")
plt.xlabel("Email Strategy ID")
plt.ylabel("Number of clicks")
axes = plt.gca()
axes.yaxis.grid()
plt.show()
```



We observe that clicked Email Strategies follow a gaussian distribution, and that O-type strategies are overwhelmingly the ones most clicked.

In [62]:

```
D = subjectD
D = dict(sorted(D.items(), key=lambda item: item[1]))
plt.figure(figsize=(17,8))
plt.bar(range(len(D)), list(D.values()), align='center')
plt.xticks(range(len(D)), list(D.keys()),fontsize = 14,rotation='vertical')
plt.title("Email Subject Lines & Clicks")
plt.xlabel("Email Strategy ID")
plt.ylabel("Number of clicks")
axes = plt.gca()
axes.yaxis.grid()
plt.show()
```



Finally, we observe a high correlation between punctuation (question / exclamation marks) in subject lines and the click rate. Upon closer analysis, subject lines containing more collaborative language and inclusive pronouns, (i.e. we, we've, let's) have significantly higher click rates than subject lines containing impersonal and/ or isolating language (i.e. you, your, alerts, notifications).

Data Cleaning

Before implementing Machine Learning, we have to clean our data in order to remove statistical noise, take into account the pull from outliers,
and format data with respect to standards machine learning models require before training can occur.

Step 1) Drop Irrelevant Columns & No Value Entries

In [46]:

```
df = df.drop(['clientid'
              ,'PlanCreatedDate'
              ,'TotalPayment'
              ,'Delivered'
              ,'Bounced','Spam'
              ,'EmailType'
              ,'FailedLogin'
              ,'1stPaymentAmount'
              ,'1stPaymentDate'
              ,'LastPaymentDate'
              ,'TotalTermInMonths'
              ,'SentDate'
              ,'PlaceDate'
              ,'cdn'
              , 'bannertext'
              ,'emailaddress'], axis=1).dropna()
```

Step 2) Clean Features using Binary Encoding

Binary Encoding is a method that turns categorical data into numerical data that machine learning models can understand.
For example, if a column called temperature contained the values 'hot', 'warm', 'cool', and 'cold', Binary Encoding would turn
the labels into their corresponding binary strings (eg. hot → 001, warm → 010, cool → 011, cold → 100), then split the strings into three columns.
In this case, we will binary encode the following features:
'emailtemplateid'
'emailstrategyid'
'mastertemplate'
'name'
'subjectline'
'EmailTreatmentStrategy'

In [47]:

```
featurestoBinarize = ['emailtemplateid','emailstrategyid','mastertemplate','name','subje
ctline','EmailTreatmentStrategy']

result = df

for feature in featurestoBinarize:
    eid = pd.DataFrame(df[feature])
    eid.columns =[feature]
    encoder = ce.BinaryEncoder(cols = [feature])
    eid_bin = encoder.fit_transform(eid[feature])
    eid = pd.concat([eid,eid_bin],axis = 1)
    result = pd.concat([result,eid_bin], axis=1)
    result = result.drop([feature], axis = 1)

# Map labels to 0 & 1 ints
result = result.replace({'AccountType': {'Digital + Calls': 1,  'Digital Only': 0}})
result = result.replace({'ContainsOffer': {'YES': 1,  'NO': 0}})
result = result.replace({'ContainsOfferTable': {'YES': 1, 'YES ':1,  'NO': 0}})
result = result.replace({'ContainsSettlement': {'YES': 1,  'NO': 0}})
result = result.replace({'Opened': {2: 1}})
result = result.replace({'Clicked': {2: 1}})

# Drop columns with a 0 in the column name
result = result[result.columns.drop(list(result.filter(regex='0')))]
```

Store cleaned data in Result dataframe

In [48]:

```
result.head()
```

Out[48]:

	acctid	AccountType	Opened	Clicked	SuccessfulLogin	ContainsOffer	ContainsOfferTable	ContainsSettlement	emailte
0	48427094	0	0	0	0	1	1	0	
1	48427189	1	0	0	1	1	1	0	
2	48427197	1	0	0	0	1	1	0	
3	48368808	0	1	1	1	1	1	0	
4	48369342	0	0	0	0	1	1	0	

5 rows × 38 columns

Step 3) Join the two databases on the shared Account Id feature

In [49]:

```
ad = df['Opened'].to_frame()
ad['acctid'] = df['acctid']
wlAccounts = pd.merge(wlAccounts, ad, on='acctid')
wlAccounts = wlAccounts.drop(['acctid'],axis=1)

wlAccounts
```

Out[49]:

	OasisEligible	CareEligible	ReInEligible	Decile	PreCharge	Opened
0	YES	NO	NO	NaN	Y	0
1	YES	NO	NO	10.0	N	0
2	YES	NO	NO	2.0	N	0
3	YES	NO	NO	NaN	Y	0
4	YES	NO	NO	NaN	Y	0
...	...	...	...	...	...	...
225850	YES	NO	NO	NaN	Y	0
225851	YES	NO	NO	NaN	Y	0
225852	YES	NO	NO	NaN	Y	0
225853	YES	NO	NO	NaN	Y	0
225854	YES	YES	NO	NaN	Y	0

225855 rows × 6 columns

Step 4) Convert YES / NOs into 1 / 0s

As previously mentioned, part of the cleaning process is turning label data into numerical data that machine learning models can understand.

In [52]:

```
wlAccounts = wlAccounts.dropna()
wlAccounts = wlAccounts.replace({'OasisEligible': {'YES': 1,  'NO': 0}})
wlAccounts = wlAccounts.replace({'CareEligible': {'YES': 1,  'NO': 0}})
wlAccounts = wlAccounts.replace({'ReInEligible': {'YES': 1,  'NO': 0}})
wlAccounts = wlAccounts.replace({'PreCharge': {'Y': 1,  'N': 0}})
wlAccounts = wlAccounts.replace({'Opened': {0:'Unopened',  1:'Opened',  2:'Opened'}})
```

```
wlAccounts = wlAccounts.rename(columns={'Opened':'target'})
wlAccounts
```

Out[52]:

	OasisEligible	CareEligible	ReInEligible	Decile	PreCharge	target
1	1	0	0	10.0	0	0
2	1	0	0	2.0	0	0
13	1	0	0	6.0	0	1
14	1	0	0	2.0	0	1
15	1	0	0	6.0	0	0
...	...	...	...	...	...	...
225760	1	0	0	6.0	0	0
225762	1	0	0	9.0	0	0
225781	1	0	0	8.0	0	1
225795	1	1	0	2.0	1	0
225797	1	0	0	4.0	0	1

198991 rows × 6 columns

Step 5) Scale Values down to Standard Length to Remove Statistical Variance

By scaling down each feature's entries to restrict their domain between [-1,1], our machine learning model can better process the data if they "weigh" equally against one another. This way, the model has no initial bias towards numerically higher or lower features.
For example, if one column called 'salary' contained values in the thousands (i.e. 65,000, 80,000), and another column called 'weekly hours' contained smaller values (i.e. 40, 45), a machine learning model may consider the salary column of a higher weight prior to standardized scaling.

In [23]:

```
features = ['OasisEligible', 'CareEligible', 'ReInEligible', 'Decile', 'PreCharge']
# Separating out the features
x = wlAccounts.loc[:, features].values
# Separating out the target
y = wlAccounts.loc[:,['target']].values
# Standardizing the features
x = StandardScaler().fit_transform(x)
x
```

Out[23]:

```
array([[ 0.28346318, -0.08338371, -0.21249408,  1.09324321, -0.09924864],
       [ 0.28346318, -0.08338371, -0.21249408, -1.45846495, -0.09924864],
       [ 0.28346318, -0.08338371, -0.21249408, -0.18261087, -0.09924864],
       ...,
       [ 0.28346318, -0.08338371, -0.21249408,  0.45531617, -0.09924864],
       [ 0.28346318, 11.99275011, -0.21249408, -1.45846495, 10.07570519],
       [ 0.28346318, -0.08338371, -0.21249408, -0.82053791, -0.09924864]])
```

Step 6) Apply Principal Component Analysis (PCA) to Reduce Dimensionality

Principal Component Analysis is a dimensionality-reduction method that is often used to reduce the number of features of large datasets, by transforming a set of vectors into a smaller set that retains the maximal amount of information about (i.e. "captures the essence of") the input features.
In order to graphically interpret how our five features (OasisEligible, CareEligible, ReInEligible, Decile,

PreCharge) correlate with the Opened feature, we apply PCA to transform 5 features down to 2 in order to plot them against each other.

In [24]:

```
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents, columns = ['principal component 1
', 'principal component 2'])
finalDf = pd.concat([principalDf, wlAccounts[['target']]], axis = 1).dropna()
```

Displayed below is the reduced dataset after applying PCA:

In [25]:

```
finalDf
```

Out[25]:

	principal component 1	principal component 2	target
1	-0.006931	0.939599	Unopened
2	-0.114725	0.075859	Unopened
13	0.020018	1.155533	Opened
14	0.020018	1.155533	Opened
15	0.020018	1.155533	Unopened
...	...	...	...
198986	-0.114725	0.075859	Unopened
198987	-0.195571	-0.571945	Opened
198988	-0.168622	-0.356011	Unopened
198989	15.668968	-0.100719	Opened
198990	-0.060828	0.507729	Opened

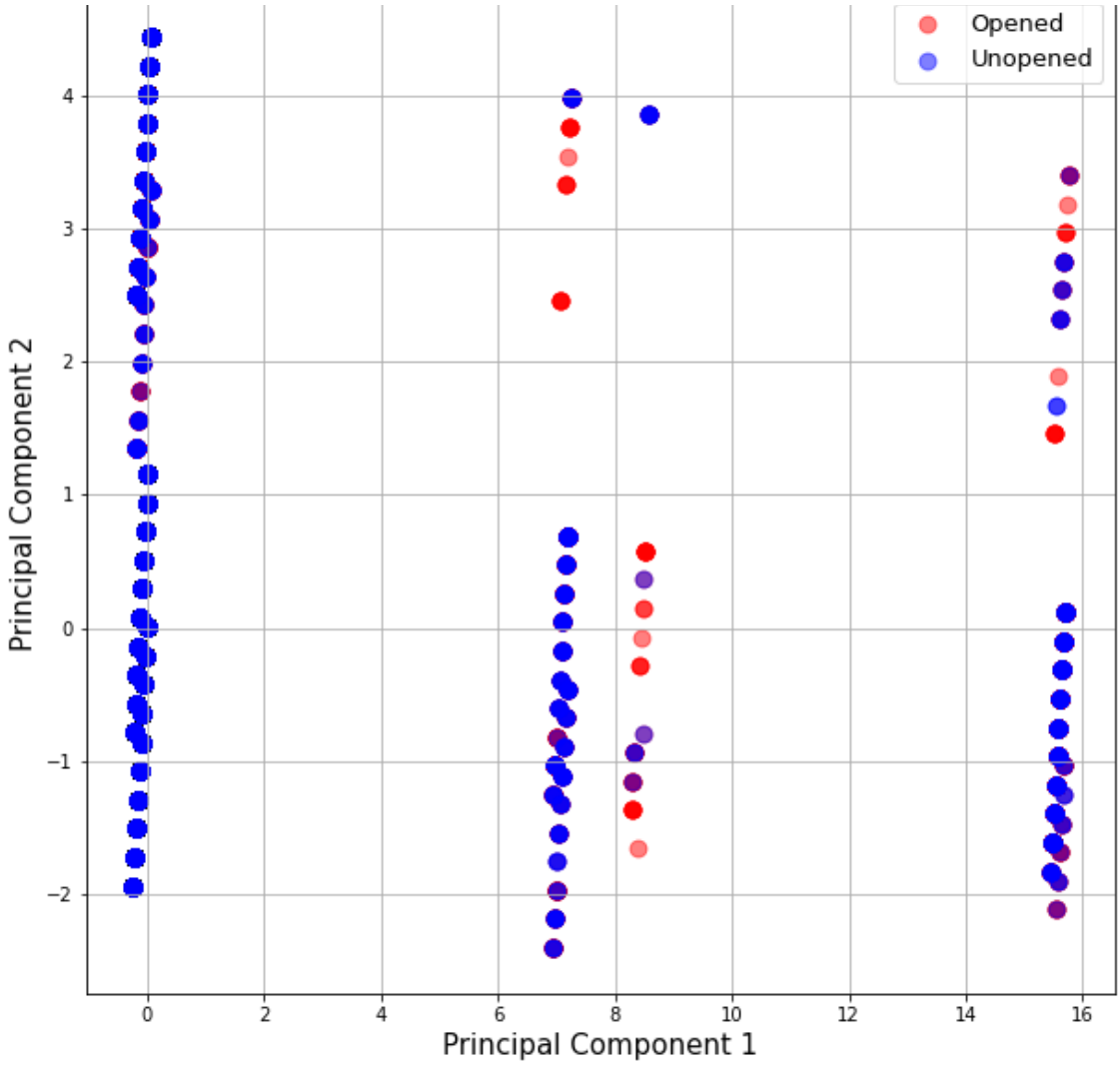
176725 rows × 3 columns

Step 7) Clustering

After properly cleaning and reducing the dataset into a two-dimensional matrix, we can plot the two princial components against each other and color the opened & unopened emails accordingly:

In [26]:

```
fig = plt.figure(figsize = (10,10))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 Component PCA', fontsize = 20)
targets = ['Unopened', 'Opened']
targets1 = ['Opened', 'Unopened']
colors = ['r','b']
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['target'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
               , finalDf.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 80
               , alpha = 0.5)
ax.legend(targets1,prop={'size': 13})
ax.grid()
```

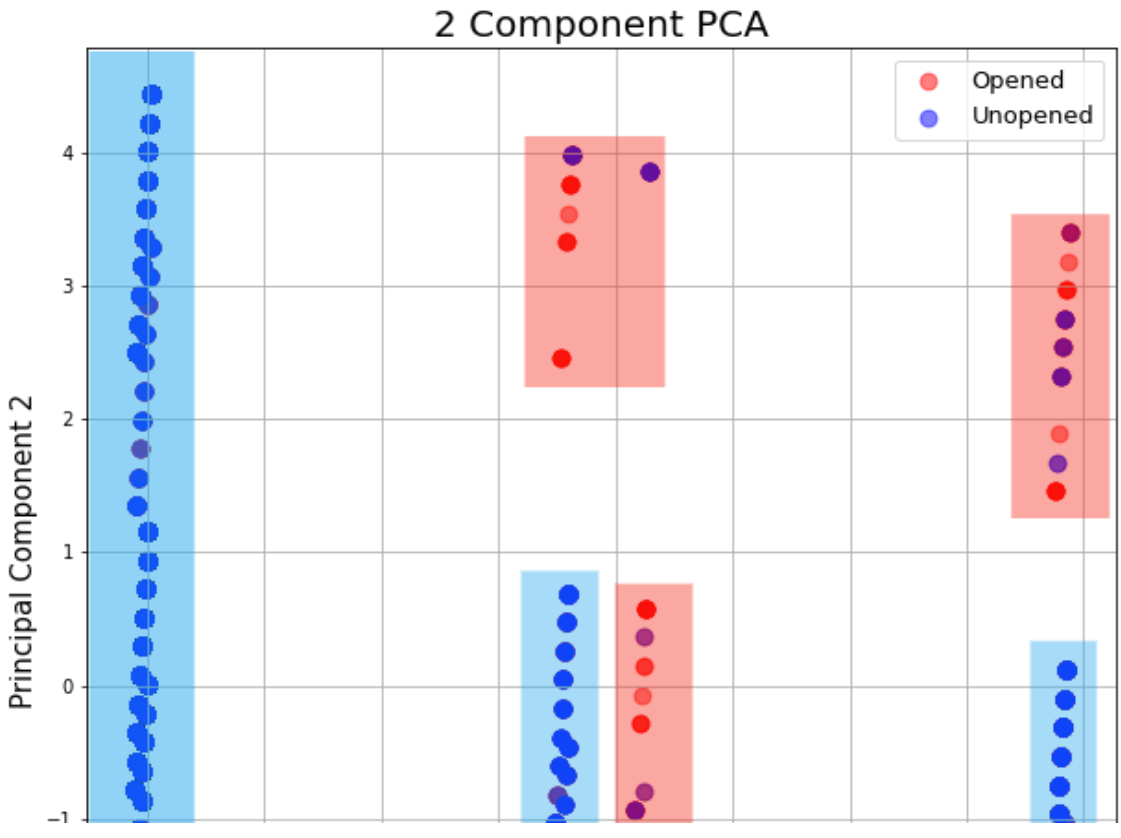


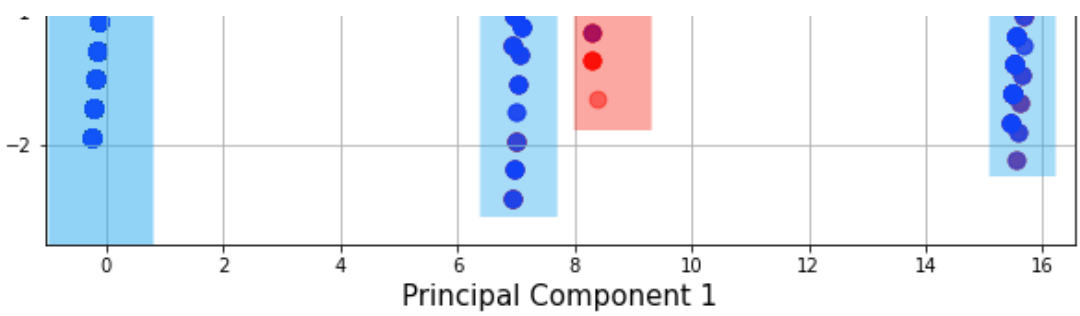
If we inutitively paint in the six lines our data inhabits, we can create opened and unopened clusters that can be used to predict where new data points (future emails) would fall into.

In [66]:

```
Image(filename=r"C:\Users\LG735620\Desktop\pca colored.png")
```

Out[66]:





Results

Using this approach, if we recluster a different quarter of a million entries from the same two tables, we achieve a binary classification accuracy of approximately ~88% for predicting email opens.

Conclusion

In summary, we visually explored our email data, discovered a strong correlation between subject lines containing collaborative language and the click through rate, cleaned our data, and clustered open and unopened entries into six regions. These six clusters are validated to have an accuracy of 88% in predicting which emails will be opened.