Predictive Modelling

Spyros Samothrakis
Lecturer/Assistant Director@IADS
University of Essex

January 29, 2018

Optimal Projects

- Let's have a look at some projects I consider superb
- Could potentially guide your work

Predictive Modelling

- We will change paradigm now and focus on one form of predictive modelling
  - Supervised learning
- "Can I predict a quantity if I know other relevant quantities"
  - Can I predict the weather if know historical data, time of the year etc
  - Can I predict if a car will break down given how old it is?
  - Facebook likes on a page?

How good are these predictions?

- Quality of learning
  - Bias - Variance decomposition
  - Measuring the quality of a supervised learning algorithm
  - Getting error estimates on predictions
- Another name for "applied machine learning"
  - Statistics, Machine Learning, Data Mining, Data Science

Scikit-learn

- This is not a machine learning module, we have more of a "birds-eye" view of the algorithms involved
  - We will see them as objects that have various interesting properties
  - You still need to understand what they do more or less
- You will need to be able to use scikit-learn to create new models
- Essential part of the Data Science toolbox
- Golden trinity alongside Pandas and Numpy

Modelling data

The example data I am going to use today comes from:

Moro, Sérgio, Paulo Rita, and Bernardo Vala. "Predicting social media performance metrics and evaluation of the impact on brand building: A data mining approach." Journal of Business Research 69.9 (2016): 3341-3351.

Around 500 publicly available instances

We are trying to predict the performance of a new post

Given a list of samples

	Category	Page total likes	Type	Post Month	Post Hour	Post Weekday	Paid
0	2	139441	Photo	12	3	4	0.0
1	2	139441	Status	12	10	3	0.0
2	3	139441	Photo	12	3	3	0.0
3	2	139441	Photo	12	10	2	1.0
4	2	139441	Photo	12	3	2	0.0

	like	share	Total Interactions
0	79.0	17.0	100
1	130.0	29.0	164
2	66.0	14.0	80
3	1572.0	147.0	1777
4	325.0	49.0	393

Loading the data

```
df = pd.read_csv("./dataset_Facebook.csv", delimiter = ";")
df.head() # just prints top 5 columns if you are in ipython

features = ["Category",
            "Page total likes",
            "Type",
            "Post Month",
            "Post Hour",
            "Post Weekday",
            "Paid"]


outcomes=  ["Lifetime Post Total Reach",
            "Lifetime Post Total Impressions",
            "Lifetime Engaged Users",
            "Lifetime Post Consumers",
            "Lifetime Post Consumptions",
            "Lifetime Post Impressions by people who have liked your Page",
            "Lifetime Post reach by people who like your Page",
            "Lifetime People who have liked your Page and engaged with your post",
            "comment",
            "like",
            "share",
            "Total Interactions"]
```


Cleaning up

scikit-learn does not accept string formatted data

```
df = df.dropna()

outcomes_of_interest = ["Lifetime Post Consumers", "likes"]

df[["Type"]] = df[["Type"]].apply(LabelEncoder().fit_transform)

X_df = df[features].copy()
y_df = df[outcomes_of_interest].copy()

X_df.head()
```

	Category	Page total likes	Type	Post Month	Post Hour	Post Weekday	Paid
0	2	139441	1	12	3	4	0.0
1	2	139441	2	12	10	3	0.0
2	3	139441	1	12	3	3	0.0
3	2	139441	1	12	10	2	1.0
4	2	139441	1	12	3	2	0.0

Playing with dataframes

- Make sure you make copies of the dataframe if you want to modify it but keep the original
- Otherwise it is just views on the same data
  - You will modify everything
- Note the double "[[column1, column2...]]" notation
  - If you just use [column1] you get back a Series, not a DataFrame
- We used the LabelEncoder to encode the labels
  - but how about decoding?

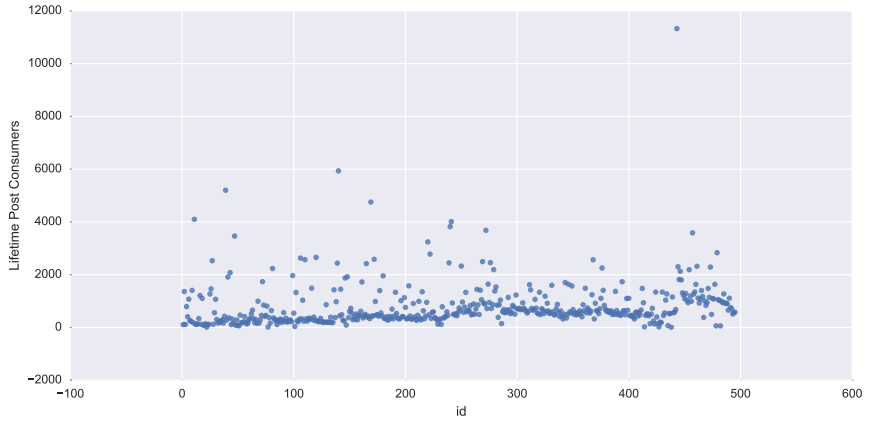
Some pandas trivia

```
# Adding a new column
y_df['id'] = range(1, len(df) + 1)
```

- If the column exists, it will replace it
- If the column does not, it will create it
- Make sure the data you are trying to insert has the same column length

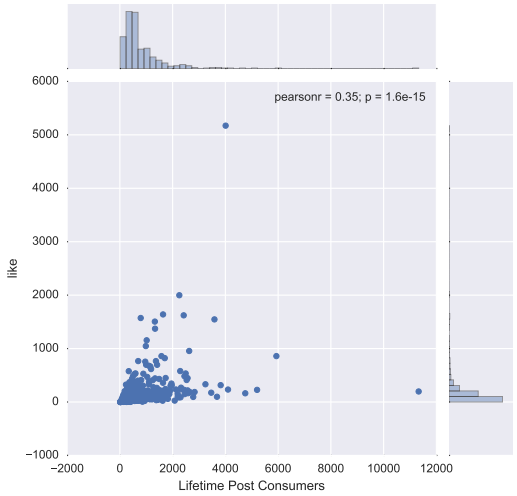
Plotting the data

```
# Notice how aspect changes the aspect ratio
sns_plot = sns.lmplot(x="id", y= attribute, data=y_df, fit_reg=False, aspect = 2)
```



Joint Plot with "likes"

```
# Notice how "aspect" changes the aspect ratio
sns_plot = sns.lmplot(x="id", y= attribute, data=y_df, fit_reg=False, aspect = 2)
```



Modelling the data

- We would like to learn some model of the data in relationship to the outcome
- That is, if someone gives as a row of features we should be able to predict the corresponding outcome
- The word "model" is overloaded, it is used for many different things
  - Use depends on context

Decision Trees and Random Forests

- For classification and regression we are going to use CART Trees
- A tree creates a function between the features and an output attribute
- When this output is real-valued the procedure is called "regression"
- When you are predicting a certain type (e.g. "apple" vs "oranges") it is called classification
- If you fit multiple decision trees on different bootstraps of the data (and features) and get the mean you have a random forest
  - Ignore this for the moment
  - Just think of Random forests as better at modelling than decision trees

Prediction

- Let's build a tree and fit on the data

```
X = X_df.values
y = y_df.values.T[0]
y = (y-y.min())/(y.max() - y.min())


clf.fit(X,y)

print mse(y,clf.predict(X))
```


Mean squared Error

- Reality is $f(x)$
- Our model is $\hat{f}(x)$ (e.g. a decision tree)
- Sample from the model are $\{y_0 ... y_N\}$
  - $MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{f}(x_i)\right)^2$
- For every possible sample
  - $E\left[\left(y - \hat{f}(x)\right)^2\right]$
- 1.5083614548e-05
- Is this number meaningful? vs what?
- Let's build a tree with fictional dice rolls
- Let's roll two dice
  - See if we can have a tree there...
  - This is known as overfitting - we are not learning anything of importance

Bias-variance decomposition

- What does this error represent?
- One way of understanding the quality of a regressor is to use the bias-variance decomposition of the error
- The noise is the lower bound on performance
- Bias is the error you expect because reality and your model are different
- Variance is the error you expect due to random noise and only seeing a sample of your data

A bit more formally

$$E\left[(y - \hat{f}(x))^2\right] = \left(E[\hat{f}(x)] - f(x)\right)^2 + E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right] + \sigma^2$$

$$E\left[(y - \hat{f}(x))^2\right] = Bias^2 + Variance + Noise^2$$

$$Bias^2 = \left(E[\hat{f}(x)] - f(x)\right)^2$$

$$Variance = E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right] = Var[\hat{f}(x)]$$

$$Noise = \sigma^2$$

Bias

- A high bias model is strongly opinionated vs the data
  - A constant function (predicting "0" all the time)
  - A linear function on very non-linear data
- A low bias fits the data closely
  - A model that was created to fit the data
    - A decision tree when the data was created using rules
  - A linear function when the data is generated by a linear process

Variance

- A high variance model changes opinions about the data based on the samples it sees
  - A decision tree on very few data points
  - A very complex model on a very simple function
- A low variance model keeps is not impacted by the training data that much
  - A constant function
  - The mean
- Variance comes from randomness in the training set

The tradeoff

- We can't really measure the noise that easily
  - It's also the lower bound on our performance
- Models with high variance have low bias
- Models with high bias have low variance

Bias-variance - using the bootstrap (1)

Let's assume noise is zero, hence:

$$E\left[(y-\hat{f}(x))^2\right] = \left(E[\hat{f}(x)] - f(x)\right)^2 + E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right]$$

We don't have access to all the samples - what should we do?

Bias-variance - using the bootstrap (2)

```
n_test = 100
n_repeat = 1000

#estimator = DecisionTreeRegressor()
estimator = RandomForestRegressor()

# Compute predictions
y_predicts = np.zeros((n_repeat, len(X)))
for i in range(n_repeat):

    sample  = np.random.choice(range(len(X)),replace = True, size = len(X))

    train_ids = sample[:-n_test]
    test_ids  = sample[-n_test:]
    test_ids = np.setdiff1d(test_ids, train_ids)
    if(len(test_ids) == 0 ):
        continue

    X_train,y_train = X[train_ids], y[train_ids]
    X_test, y_test = X[test_ids], y[test_ids]

    estimator.fit(X_train, y_train)
    y_predict = estimator.predict(X_test)
    y_predicts[i,test_ids] = y_predict
```


Bias-variance - using the bootstrap (3)

```
y_bias = (y - np.mean(y_predicts, axis=0)) **2

y_error = ((y - y_predicts) **2).mean()
y_var = np.var(y_predicts, axis=0, ddof = 1)

clf_type = "Decision tree"
print("{0}: {1:.4f} (error) = {2:.4f} (bias^2) "
          "+ {3:.4f} (var)".format(clf_type,
                                   np.mean(y_error),
                                   np.mean(y_bias),
                                   np.mean(y_var)))

print("{0}: {1:.4f} ((bias^2) + (var)) = {2:.4f} (bias^2) "
          "+ {3:.4f} (var)".format(clf_type,
                                   np.mean(y_bias) + np.mean(y_var),
                                   np.mean(y_bias),
                                   np.mean(y_var)))
```


Measuring error

Decision tree: 0.0110 (error) = 0.0100 (bias^2) + 0.0010 (var)

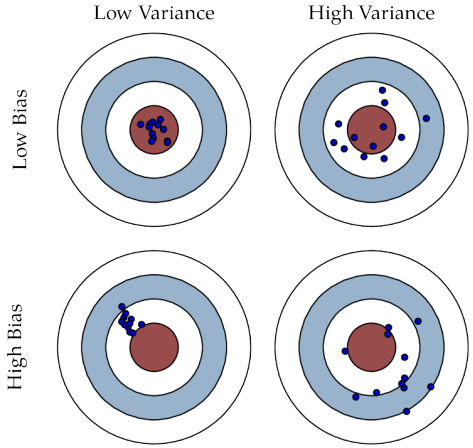
Decision tree: 0.0110 ((bias^2) + (var)) = 0.0100 (bias^2) + 0.0010 (var)

Random Forest: 0.0107 (error) = 0.0099 (bias^2) + 0.0008 (var)

Random Forest: 0.0107 ((bias^2) + (var)) = 0.0099 (bias^2) + 0.0008 (var)

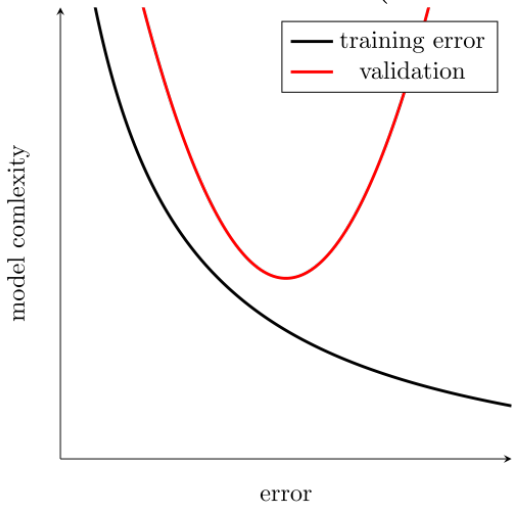
- The procedure for using the bootstrap to find better values for a model is called “Bagging”
  - Mostly minimises the variance
  - Very commonly used trick

Intuition



http://tex.stackexchange.com/questions/307117/reconstructing-the-following-bias-variance-diagram

Error vs validation (or test set)



https://github.com/MartinThoma/LaTeX-examples/tree/master/tikz/bias-variance

Cross validation (1)

- One of the most common procedures used when evaluating a model
  - We take data X,y
  - Split into 10 different random chunks
- Learn a model using 9 of them
  - Test performance on the one you did not learn on
  - Pick another 9, test on another set
- Keep on going until you test on all test subsets
- K-fold cross-validation
  - There are other methods, but all related to this # Cross validation and Regression

Cross validation (2)

It is largely automated on scikit-learn

```
clf = RandomForestRegressor(n_estimators = 1000,max_depth = 2)
dummy_clf = DummyRegressor()
scores = cross_val_score(clf, X, y, cv=10,scoring = make_scorer(mse))
dummy_scores = cross_val_score(dummy_clf, X, y, cv=10, scoring = make_scorer(mse))

print("MSE: %0.8f (+/- %0.8f)" % (scores.mean(), scores.std()))
print("Dummy MSE: %0.8f (+/- %0.8f)" % (dummy_scores.mean(), dummy_scores.std()))
```

MSE: 0.00699996 (+/- 0.00748646) Dummy MSE: 0.00616020 (+/- 0.00521769)

What can we say about our model?

- It is bad
- We are worse than just predicting the average!
- Hence we can't really say much about if there is any relationship between our variables of interest
  - You can't prove a negative
  - Maybe our regressor is not good enough?
- We are not seeing any predictive effect here

Ridge Regression

Maybe we can get better results with a different regressor?

```
cat_features = ["Category",
            "Type",
            "Paid"]


X_df = pd.get_dummies(X_df, cat_columns = features)
```

Here is how "category" is converted to dummy

	Category_1	Category_2	Category_3
0	0.0	1.0	0.0
1	0.0	1.0	0.0
2	0.0	0.0	1.0
3	0.0	1.0	0.0
4	0.0	1.0	0.0

BayesianRidge Regression

```
from sklearn.linear_model import  BayesianRidge
clf = BayesianRidge(normalize = True)
dummy_clf = DummyRegressor()
scores = cross_val_score(clf, X, y, cv=10,scoring = make_scorer(mse))
dummy_scores = cross_val_score(dummy_clf, X, y, cv=10, scoring = make_scorer(mse))

print("MSE: %0.8f (+/- %0.8f)" % (scores.mean(), scores.std()))
print("Dummy MSE: %0.8f (+/- %0.8f)" % (dummy_scores.mean(), dummy_scores.std()))
```

MSE: 0.00479575 (+/- 0.00511744) Dummy MSE: 0.00616020 (+/- 0.00521769

Not great (at all!), but better

Binning samples

- Maybe we can try to create classes out of the data

```
outcomes_of_interest = ["Lifetime Post Consumers", "like"]
n_bins = 20

X_df = df[features].copy()
y_df = df[outcomes_of_interest].copy()

bins =  pd.qcut(y_df[outcomes_of_interest[0]].values,n_bins)
y_df = df[outcomes_of_interest].copy()
y_df[outcomes_of_interest[0]] = bins
y_df[outcomes_of_interest] = y_df[outcomes_of_interest].apply(LabelEncoder().fit_transform)
```


Metrics for classification

- Each row is now assigned to a class of $y_i \in 0..20$
- Accuracy is the obvious one
  - $accuracy = \frac{1}{N}\sum_{i=0}^{N-1} y_i = \hat{f}(x)$
  - The higher the accuracy the better
- What if the dataset is unbalanced - how informative is accuracy then?
- There are multiple metric functions
  - Use the one appropriate for your problem

Now with the classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier

clf = ExtraTreesClassifier(n_estimators = 1000,max_depth = 4)

dummy_clf = DummyClassifier()
scores = cross_val_score(clf, X, y, cv=10,scoring = make_scorer(acc))
dummy_scores = cross_val_score(dummy_clf, X, y, cv=10, scoring = make_scorer(acc))

print("ACC: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()))
print("Dummy ACC: %0.2f (+/- %0.2f)" % (dummy_scores.mean(), dummy_scores.std()))
```

ACC: 0.18 (+/- 0.06) Dummy ACC: 0.05

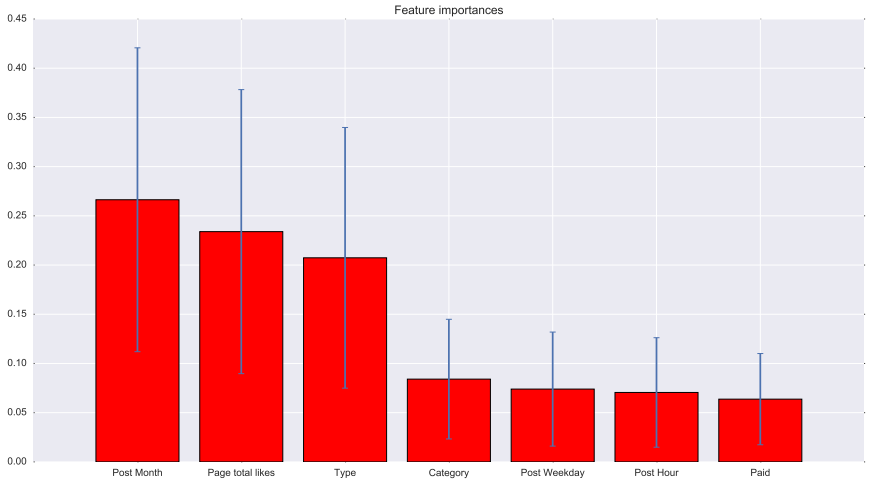
Generate feature importances

```
importances = clf.feature_importances_
std = np.std([tree.feature_importances_ for tree in clf.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

print("Feature ranking:")
for f in range(X.shape[1]):
    print("%d. %s (%f)" % (f + 1, features[indices[f]],  importances[indices[f]]))

# Plot the feature importances of the forest
fig = plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), np.array(features)[indices])
plt.xlim([-1, X.shape[1]])
fig.set_size_inches(15,8)
axes = plt.gca()
axes.set_ylim([0,None])
```


Feature importances



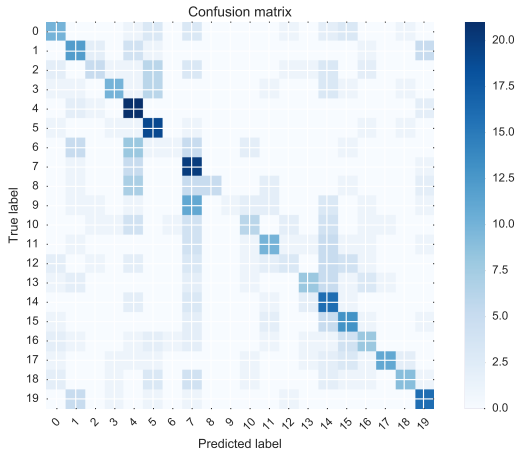
Generate a confusion matrix

```
###... code

# Compute confusion matrix
y_pred = clf.predict(X)
cnf_matrix = confusion_matrix(y, y_pred)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=range(len(set(y))), normalize = True,
                      title='Confusion matrix')
```

Confusion Matrix?



Conclusion

- We have just touched the subject
- More in the labs
- We will revisit some of the concepts later on
- You should (after the labs) be able to train a basic model
- If you are to optimise hyperparameters, you need to do nested cross-validation
  - Hyperparameters are things like tree size, depth etc.
  - Again, we will see this later on