Scikit-learn requires numeric data and no missing values

We need to pre-process data to ensure data validity

Dealing with categorical features

Convert to binary values and use dummy variables

get_dummies is important

```
import pandas as pd
music_df = pd.read_csv('music.csv')
music_dummies = pd.get_dummies(music_df["genre"], drop_first=True)

music_dummies = pd.concat([music_df, music_dummies], axis=1)
music_dummies = music_dummies.drop("genre", axis=1)

from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LinearRegression
X = music_dummies.drop("popularity", axis=1).values
y = music_dummies["popularity"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_State=42)
kf = KFold(n_splits=5, shuffle=True, random_State=42)
linreg = LinearRegression()
linreg_cv = cross_val_score(linreg, X_train, y_train, cv=kf,
scoring="neg_mean_squared_error")

print(np.sqrt(-linreg_cv))
```

Handling Missing Data

```
import pandas as pd
music_df = music_df.dropna(subset=
["genre","popularity","loudness","liveness","tempo"])
```

You could impute missing data, that is replacing missing data with educated guesses using the most frequent value, the mode

Must split data first, to avoid data leakage

Categorical features: Use the most frequent value (Strategy="most frequent")
Numerical features: Use the mean (default strategy)

```
from sklearn.impute import SimpleImputer
X_cat = music_df["genre"].values.reshape(-1,1)
X_num = music_df.drop(["genre", "popularity"], axis=1).values
y = music_df["popularity"].values
X_train_cat, X_test_cat, y_train_cat, y_test_cat = train_test_split(X_cat, y,
test_size=0.2, random_state=12)
X_train_num, X_test_num, y_train, y_test = train_test_split(X_num, y,
test_size=0.2, random_state=12)
imp_cat = SimpleImputer(strategy="most_frequent")
X_train_cat = imp_cat.fit_transform(X_train_cat)
X_test_cat = imp_cat.transform(X_test_cat)

imp_num = SimpleImputer()
X_train_num = imp_num.fit_transform(X_train_num)
X_test_num = imp_num.transform(X_test_num)
X_train = np.append(X_train_num, X_train_cat, axis=1)
X_test = np.append(X_test_num, X_tes_cat, axis=1)
```

Imputers are known as transformers.

We could also impute with a pipeline.

```
from sklearn.pipeline import Pipeline
music_df = music_df.dropna(subset=
["genre","popularity","loudness","liveness","tempo"])
music_df["genre"] = np.where(music_df["genre"] == "Rock", 1, 0)
X = music_df.drop("genre", axis=1).values
y = music_df["genre"].values

steps = [("imputation", SimpleImputer()), ("logistic_regression",
LogisticRegression())]
pipeline = Pipeline(steps)
X_train, X_test, y_Train, y_test = train_test_split(X, y, test_size=0.3,
random_State=42)
pipeline.fit(X_train, y_train)
pipeline.score(X_test, y_test)
```

Centering and scaling

df.describe sees count, mean, std, min, max

features on larger scales are hard to use, because many models use some form of distance

We can normalise or standardise data (scaling and centering)

We can scale data by subtracting by the mean and dividing by the variance so that all features are centered around zero and have a variance of one, this is **standardization**

We can also subtract the minimum and divide by the range

We can also normalize so that data ranges from -1 to 1

```
from sklearn.preprocessing import StandardScaler
X = music_df.drop("genre", axis=1).values
y = music_df["genre"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
print(np.mean(X), np.std(X))
print(np.mean(X_train_scaled), np.std(X_train_scaled))
```

or pipeline

```
steps = [('scaler', StandardScaler()), 'knn',
KNeighborsClassifier(n_neighbors=6)]
pipeline = Pipeline(Steps)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=21)
knn_scaled = pipeline.fit(X_train, y_train)
y_pred = knn_scaled.predict(X_test)
print(knn_scaled.score(X_test, y_test))
```

or unscaled

```
knn_unscaled = KNeighborsClassifier(n_neighbors=6).fit(X_train, y_train)
print(knn_unscaled_score(X_test, y_test))
```

```
from sklearn.model_selection import GridSearchCV
steps = [('scaler', StandardScaler()), 'knn', KNeighborsClassifier()]
pipeline = Pipeline(steps)
parameters = {'knn__n_neighbors": np.arange(1,50)}
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=21)
cv = GridSearchCV(pipeline, param_grid=parameters)
cv.fit(X_train, y_train)
y_pred = cv.predict(X_test)

print(cv.best_score_)
print(cv.best_params_)
```

Evaluating multiple models

Principles:

1. Size of the dataset - fewer features = simpler model

2. Interpretability - explaining to stakeholders (e.g. linreg)
3. Flexibility - flexible models = fewer assumptions (e.g. knn)

All regression can be evaluated:

- RMSE
- R-squared

All classification model performance:

- Accuracy
- Confusion matrix
- Precision, recall, F1- score
- ROC AUC

Performance of some models are affected by scaling data:

- knn
- linreg (incl. ridge and lasso)
- logreg
- artifical NNs